MX

macromedia®
FLASH™MX
2004

Using Components

Acknowledgments

Director: Erick Vera

Project Management: Julee Burdekin, Erick Vera

Writing: Jay Armstrong, Jody Bleyle, Mary Burger, Francis Cheng, Jen deHaan, Stephanie Gowin, Phillip Heinz, Shimul Rahim, Samuel R. Neff

Managing Editor: Rosana Francescato

Editing: Mary Ferguson, Mary Kraemer, Noreen Maher, Antonio Padial, Lisa Stanziano, Anne Szabla

Production Management: Patrice O'Neill

Media Design and Production: Adam Barnett, Christopher Basmajian, Aaron Begley, John Francis

Second Edition: June 2004

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

CONTENTS

INTRODUCTION
Getting Started with Components

Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 are the professional standard authoring tools for producing high-impact web experiences. Components are the building blocks for the Rich Internet Applications that provide those experiences. A component is a movie clip with parameters that are set during authoring in Macromedia Flash, and with ActionScript methods, properties, and events that allow you to customize the component at runtime. Components are designed to allow developers to reuse and share code, and to encapsulate complex functionality that designers can use and customize without using ActionScript.

Components are built on version 2 of the Macromedia Component Architecture, which allows you to easily and quickly build robust applications with a consistent appearance and behavior. This book describes how to build applications with version 2 components and describes each component's application programming interface (API). It includes usage scenarios and procedural samples for using the Flash MX 2004 or Flash MX Professional 2004 version 2 components, as well as descriptions of the component APIs, in alphabetical order.

You can use components created by Macromedia, download components created by other developers, or create your own components.

This chapter contains the following sections:

Intended audience

This book is for developers who are building Flash MX 2004 or Flash MX Professional 2004 applications and want to use components to speed development. You should already be familiar with developing applications in Flash and writing ActionScript.

If you are less experienced with writing ActionScript, you can add components to a document, set their parameters in the Property inspector or Component inspector, and use the Behaviors panel to handle their events. For example, you could attach a Go To Web Page behavior to a Button component that opens a URL in a web browser when the button is clicked without writing any ActionScript code.

If you are a programmer who wants to create more robust applications, you can create components dynamically, use ActionScript to set properties and call methods at runtime, and use the listener event model to handle events.

For more information, see Chapter 3, "Working with Components," on page 43.

System requirements

Macromedia components do not have any system requirements in addition to Flash MX 2004 or Flash MX Professional 2004.

Any SWF file that uses version 2 components must be viewed with Flash Player 6 (6.0.79.0) or later.

About the documentation

This document explains the details of using components to develop Flash applications. It assumes that you have general knowledge of Macromedia Flash and ActionScript. Specific documentation about Flash and related products is available separately.

This document is available as a PDF file and as online help. To view the online help, start Flash and select Help > Using Components.

For information about Macromedia Flash, see the following documents:

- *Getting Started with Flash* (or Getting Started Help)
- *Using Flash* (or Using Flash Help)
- *Using ActionScript in Flash* (or Using ActionScript Help)
- *Flash ActionScript Language Reference* (or Flash ActionScript Language Reference Help)

Typographical conventions

The following typographical conventions are used in this book:

- *Italic font* indicates a value that should be replaced (for example, in a folder path).
- `Code font` indicates ActionScript code.
- *`Code font italic`* indicates a code item that should be replaced (for example, an ActionScript parameter).
- **Bold font** indicates a value that you enter.

Note: Bold font is not the same as the font used for run-in headings. Run-in heading font is used as an alternative to a bullet.

Terms used in this manual

The following terms are used in this manual:

at runtime When the code is running in Flash Player.

while authoring While you are working in the Flash authoring environment.

Additional resources

For the latest information on Flash, plus advice from expert users, advanced topics, examples, tips, and other updates, see the Macromedia DevNet website at www.macromedia.com/devnet, which is updated regularly. Check the website often for the latest news on Flash and how to get the most out of the program.

For TechNotes, documentation updates, and links to additional resources in the Flash Community, see the Macromedia Flash Support Center at www.macromedia.com/support/flash.

For detailed information on ActionScript terms, syntax, and usage, see *Using ActionScript in Flash* and *Flash ActionScript Language Reference*.

For an introduction to using components, see the Macromedia On Demand Seminar, Flash MX 2004 Family: Using UI Components at www.macromedia.com/macromedia/events/online/ondemand/index.html.

CHAPTER 1
About Components

Components are movie clips with parameters that allow you to modify their appearance and behavior. A component can be a simple user interface control, such as a radio button or a check box, or it can contain content, such as a scroll pane; a component can also be non-visual, like the FocusManager that allows you to control which object receives focus in an application.

Components enable anyone to build complex Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 applications, even if they don't have an advanced understanding of ActionScript. Rather than creating custom buttons, combo boxes, and lists, you can drag these components from the Components panel to add functionality to your applications. You can also easily customize the look and feel of components to suit your design needs.

Components are built on version 2 of the Macromedia Component Architecture, which allows you to easily and quickly build robust applications with a consistent appearance and behavior. The version 2 architecture includes classes on which all components are based, styles and skins mechanisms that allow you to customize component appearance, a broadcaster/listener event model, depth and focus management, accessibility implementation, and more.

Each component has predefined parameters that you can set while authoring in Flash. Each component also has a unique set of ActionScript methods, properties, and events, also called an *API* (application programming interface), that allows you to set parameters and additional options at runtime.

Flash MX 2004 and Flash MX Professional 2004 include many new Flash components and several new versions of components that were included in Flash MX. For a complete list of components included with Flash MX 2004 and Flash MX Professional 2004, see "Installing components" on page 12. You can also download components built by members of the Flash community at the Macromedia Exchange at www.macromedia.com/cfusion/exchange/index.cfm.

This chapter contains the following sections:

Installing components

A set of Macromedia components is already installed when you launch Flash MX 2004 or Flash MX Professional 2004 for the first time. You can view them in the Components panel.

Flash MX 2004 includes the following components:

- Button component
- CheckBox component
- ComboBox component
- Label component
- List component
- Loader component
- NumericStepper component
- ProgressBar component
- RadioButton component
- ScrollPane component
- TextArea component
- TextInput component
- Window component

Flash MX Professional 2004 includes the Flash MX 2004 components and the following additional components and classes:

- Accordion component (Flash Professional only)
- Alert component (Flash Professional only)
- Data binding classes (Flash Professional only)
- DateField component (Flash Professional only)
- DataGrid component (Flash Professional only)
- DataHolder component (Flash Professional only)
- DataSet component (Flash Professional only)
- DateChooser component (Flash Professional only)
- Form class (Flash Professional only)
- Media components (Flash Professional only)
- Menu component (Flash Professional only)
- MenuBar component (Flash Professional only)

- RDBMSResolver component (Flash Professional only)
- Screen class (Flash Professional only)
- Slide class (Flash Professional only)
- Tree component (Flash Professional only)
- WebServiceConnector class (Flash Professional only)
- XMLConnector component (Flash Professional only)
- XUpdateResolver component (Flash Professional only)

To verify installation of the Flash MX 2004 or Flash MX Professional 2004 components:

1. Start Flash.
2. Select Window > Development Panels > Components to open the Components panel if it isn't already open.
3. Select UI Components to expand the tree and view the installed components.

You can also download components from the Macromedia Exchange at www.macromedia.com/exchange. To install components downloaded from the Exchange, download and install the Macromedia Extension Manager at www.macromedia.com/exchange/em_download/

Any component can appear in the Components panel in Flash. Follow these steps to install components on either a Windows or Macintosh computer.

To install components on a Windows-based or a Macintosh computer:

1. Quit Flash.
2. Place the SWC or FLA file containing the component in the following folder on your hard disk:
   - \Program Files\Macromedia\Flash MX 2004\<language>\Configuration\Components (Windows)
   - HD/Applications/Macromedia Flash MX 2004/Configuration/Components (Macintosh)
3. Open Flash.
4. Select Window > Development Panels > Components to view the component in the Components panel if it isn't already open.

Where component files are stored

Flash components are stored in the application-level Configuration folder.

Note: For information about these folders, see "Configuration folders installed with Flash" in *Using Flash*.

Components are installed in the following locations:

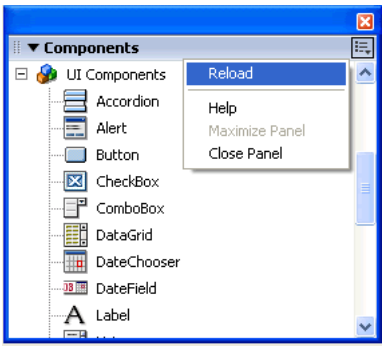
- Windows 2000 or Windows XP: C:\Program Files\Macromedia\Flash MX 2004*language*\Configuration\Components
- Mac OS X: HD/Applications/Macromedia Flash MX 2004/Configuration/Components

The source ActionScript files for components are located in the mx subfolder of the First Run folder.

If you've added components, you'll need to refresh the Components panel.

To refresh the contents of the Components panel:

- Select Reload from the Components panel menu.



To remove a component from the Components panel:

- Remove the MXP or FLA file from the Configuration folder.

Benefits of using components

Components enable the separation of coding and design. They also allow you to reuse code, either in components you create, or by downloading and installing components created by other developers.

Components allow coders to create functionality that designers can use in applications. Developers can encapsulate frequently used functionality into components and designers can customize the look and behavior of components by changing parameters in the Property inspector or the Component inspector.

Members of the Flash community can use the Macromedia Exchange at www.macromedia.com/go/exchange to exchange components. By using components, you no longer need to build each element in a complex web application from scratch. You can find the components you need and put them together in a Flash document to create a new application.

Components that are based on the version 2 architecture share core functionality such as styles, event handling, skinning, focus management, and depth management. When you add the first version 2 component to an application, there is approximately 25K added to the document that provides this core functionality. When you add additional components, that same 25K is reused for them as well, resulting in a smaller increase in size to your document than you may expect. For information about upgrading components, see "Upgrading version 1 components to version 2 architecture" on page 53.

Categories of components

Components included with Flash MX 2004 and Flash MX Professional 2004 fall into the following five categories (the locations of their ActionScript source files roughly correspond to these categories as well and are listed in parentheses):

- User interface components (mx.controls.*)

  User interface components allow you to interact with an application; for example, the RadioButton, CheckBox, and TextInput components are user interface controls.

- Data components (mx.data.*)

  Data components allow you to load and manipulate information from data sources; the WebServiceConnector and XMLConnector components are data components.

  Note: The source files for the data components aren't installed with Flash. However, some of the supporting ActionScript files are installed.

- Media components (mx.controls.*)

  Media components let you play back and control streaming media; MediaController, MediaPlayback, and MediaDisplay are media components.

- Managers (mx.managers.*)

  Managers are nonvisual components that allow you to manage a feature, such as focus or depth, in an application; the FocusManager, DepthManager, PopUpManager, StyleManager, and SystemManager components are manager components.

- Screens (mx.screens.*)

  The screens category includes the ActionScript classes that allow you to control forms and slides in Flash MX Professional 2004.

For a complete list of each category, see Chapter 6, "Components Dictionary," on page 91.

About version 2 component architecture

You can use the Property inspector or the Component inspector to change component parameters to make use of the basic functionality of components. However, if you want greater control over components, you need to use their APIs and understand a little bit about the way they were built.

Flash MX 2004 and Flash MX Professional 2004 components are built with version 2 of the Macromedia Component Architecture. Version 2 components are supported by Flash Player 6.79 and Flash Player 7. These components are not always compatible with components built using version 1 architecture (all components released before Flash MX 2004). Also, the original version 1 components are not supported by Flash Player 7. For more information, see "Upgrading version 1 components to version 2 architecture" on page 53.

Note: Flash MX UI components have been updated to work with Flash Player 7. These updated components are still based on version 1 architecture. You can download them from the Macromedia Flash Exchange at www.macromedia.com/cfusion/exchange/index.cfm#loc=en_us&view=sn106&viewName=Exchange%20Search%20Details&authorid=60639501&page=0&scrollPos=0&subcatid=0&snid=sn106&itemnumber=0&extid=1009423&catid=0.

Version 2 components are included in the Components panel as compiled clip (SWC) symbols. A compiled clip is a component movie clip whose code has been compiled. Compiled clips cannot be edited, but you can change their parameters in the Property inspector and Component inspector, just as you would with any component. For more information, see "About compiled clips and SWC files" on page 17.

Version 2 components are written in ActionScript 2.0. Each component is a class and each class is in an ActionScript package. For example, a radio button component is an instance of the RadioButton class whose package name is mx.controls. For more information about packages, see "Using packages" in *Using ActionScript in Flash*.

Most UI components built with version 2 of the Macromedia Component Architecture are subclasses of the UIObject and UIComponent classes and inherit all properties, methods, and events from those classes. Many components are also subclasses of other components. The inheritance path of each component is indicated in its entry in Chapter 6, "Components Dictionary," on page 91.

Note: The class hierarchy is also available as a GIF file (v2_Flash_component_arch.gif) in the Examples folder.

All components also use the same event model, CSS-based styles, and built-in themes and skinning mechanisms. For more information on styles and skinning, see Chapter 5, "Customizing Components," on page 67. For more information on event handling, see Chapter 3, "Working with Components," on page 43.

For a detailed explanation of the version 2 component architecture, see Chapter 7, "Creating Components," on page 915.

What's new in version 2 components

This section outlines the differences between version 1 and version 2 components from the perspective of a developer using components to build Flash applications. For detailed information about the differences between the version 1 and version 2 architectures for building components, see Chapter 7, "Creating Components," on page 915.

The Component inspector allows you to change component parameters while authoring in Macromedia Flash and Macromedia Dreamweaver. (See "Setting component parameters" on page 47.)

The listener event model allows listeners to handle events. (See Chapter 4, "Handling Component Events," on page 55.) There isn't a `clickHandler` parameter in the Property inspector, as there was in Flash MX; you must write ActionScript code to handle events.

Skin properties let you load individual skins (for example, up and down arrows or the check for a check box) at runtime. (See "About skinning components" on page 80.)

CSS-based styles allow you to create a consistent look and feel across applications. (See "Using styles to customize component color and text" on page 67.) To set a component style, use the following syntax: `componentInstance.setStyle("styleName", value)`.

Themes allow you to drag a new look from the library onto a set of components. (See "About themes" on page 77.)

The Halo theme provides a ready-made, responsive, and flexible user interface for applications. Halo is the default theme that the version 2 components use. (See "About themes" on page 77.)

Manager classes provide an easy way to handle focus and depth in a application. (See "Creating custom focus navigation" on page 50 and "Managing component depth in a document" on page 51.)

The base classes UIObject and UIComponent provide core methods, properties, and events to components that extend them. (See "UIComponent class" on page 793 and "UIObject class" on page 808.)

Packaging as a SWC file allows easy distribution and concealable code. See Chapter 7, "Creating Components," on page 915.

Built-in data binding is available through the Component inspector. For more information, see Using Flash > Data Integration.

An easily extendable class hierarchy using ActionScript 2.0 allows you to create unique namespaces, import classes as needed, and subclass easily to extend components. See Chapter 7, "Creating Components," on page 915 and *Flash ActionScript Language Reference.*

About compiled clips and SWC files

Components included with Flash MX 2004 or Flash MX Professional 2004 are not FLA files—they are compiled clips that have been packaged into SWC files. SWC is the Macromedia file format for distributing components; it contains a compiled clip, the component's ActionScript class file, and other files that describe the component. For details about SWC files, see "Exporting and distributing a component" on page 953.

When you place a SWC file in the First Run/Components folder, the component appears in the Components panel. When you add a component to the Stage from the Components panel, a compiled clip symbol is added to the library.

A compiled clip is a package of precompiled symbols and ActionScript. It's used to avoid recompiling symbols and code that aren't going to change. A movie clip can also be "compiled" in Flash and converted to a compiled clip symbol. For example, a movie clip with a lot of ActionScript code that doesn't change often could be turned into a compiled clip. The compiled clip symbol behaves just like the movie clip symbol from which it was compiled, but compiled clips appear and publish much faster than regular movie clip symbols. Compiled clips can't be edited, but they do have properties that appear in the Property inspector and the Component inspector.

To compile a movie clip symbol:

- Right-click (Windows) or Control-click (Macintosh) the movie clip in the Library panel, and then select Convert to Compiled Clip.

To export a SWC file:

- Select the movie clip in the Library panel and right-click (Windows) or Control-click (Macintosh), and then select Export SWC File.

Note: Flash MX 2004 and Flash MX Professional 2004 continue to support FLA components.

Accessibility and components

A growing requirement for web content is that it should be accessible; that is, usable for people with a variety of disabilities. Visual content in Flash applications can be made accessible to the visually impaired with the use of screen reader software, which provides a spoken audio description of the contents of the screen.

When a component is created, the author can write ActionScript that enables communication between the component and a screen reader. Then, when a developer uses components to build an application in Flash, the developer uses the Accessibility panel to configure each component instance.

Most components built by Macromedia are designed for accessibility. To find out whether a component is accessible, see its entry in Chapter 6, "Components Dictionary," on page 91. When you're building an application in Flash, you'll need to add one line of code for each component (`mx.accessibility.`*`ComponentName`*`AccImpl.enableAccessibility();`), and set the accessibility parameters in the Accessibility panel. Accessibility for components works the same way as it works for all Flash movie clips.

Most components built by Macromedia are also navigable by the keyboard. Each component's entry in Chapter 6, "Components Dictionary," indicates whether or not you can control the component with the keyboard.

CHAPTER 2
Creating an Application with Components (Flash Professional Only)

Components in Flash are prebuilt elements that you can use when creating Flash applications, to add user interface controls, data connectivity, and other functionality. Components can save you work when you're building an application, because you don't have to create all the design and functionality from scratch.

This tutorial shows how to build a Flash application using components available in Macromedia Flash MX Professional 2004, including a variety of user interface and data connectivity components. You'll learn how to work with components by using panels and other interface features in the Flash authoring environment and by using ActionScript.

About working with components

All components are listed in the Components panel. To use a component, you add an instance of the component to a Flash application.

You can add a component instance in several ways:

- To add a component instance to an application while authoring, drag the component from the Components panel onto the Stage. This also places the component in the library. You can add additional instances of the component by dragging the component from the library onto the Stage. For more information, see "The Components panel" on page 44 and "Adding components during authoring" on page 44.
- To create a component instance dynamically, first add the component to the library: drag the component from the Components panel onto the Stage and then delete the instance on the Stage (the component remains in the library). Then add ActionScript to the application to create the instance, as you would create an instance of a movie clip or other object in the library. For more information on adding components dynamically, see "Adding components at runtime with ActionScript" on page 46.

Once the component is added to the library, you can create instances either by dragging to the Stage or by writing ActionScript.

You can modify the appearance and behavior of components by setting component parameters in the authoring environment, using the Parameters tab in either the Property inspector or the Component inspector. You can also control components during runtime using ActionScript. All components have ActionScript methods, properties, and events. For more information on authoring parameters, see "Setting component parameters" on page 47.

After you build an application using components, you can update or repurpose it simply by resetting component parameters, without having to rewrite code. An application built with components can even be updated by someone who doesn't know all the code used to create it.

The components included with Flash MX 2004 and Flash MX Professional 2004 are SWC files (the Macromedia file format for components). A SWC file contains a compiled clip of the component, as well as an icon that appears in the Components panel, and other assets to create component functionality.

Compiled clips are complex symbols that are precompiled so that they are easier to work with in a Flash document. For example, both the Test Movie and the Publish procedures run faster with compiled clips, because the clips don't need to compile when the SWF file is generated.

Because components are precompiled, you cannot edit them as you would uncompiled movie clips (FLA files). You modify components by setting their parameters or by using their ActionScript methods, properties, and events.

For more general information about components, see the following topics:

- Chapter 1, "About Components"
- Chapter 3, "Working with Components"
- Chapter 6, "Components Dictionary"

About this tutorial

This tutorial is intended for intermediate Flash users who are familiar with the Flash authoring environment and have some experience with ActionScript. In the authoring environment, you should have some experience using panels, tools, the Timeline, and the library. In ActionScript, you should be familiar with writing functions, adding event listeners, and using class files.

- Build the application architecture: Add component instances and movie clips to build the application interface. This section covers adding UI and data components and setting their parameters while authoring.
- Bind components to display product information from an external source: Bind components to one another to distribute and display data from an external XML file. This section covers using the data integration features in the Flash authoring environment to bind data and UI components together.
- Add ActionScript to the main Timeline: Add ActionScript code to create interactive functionality. This section includes importing the classes for the components used in the application. Most of the code places event listeners on components to process data in response to user input.

If you are experienced with building application architecture in Flash, you may want to skip the first section of the tutorial and read the second and third sections while referring to the finished FLA file of the sample application, to learn about the procedures used to bind the components and add event listeners for data integration. (To view the finished FLA file, see the next section.)

All the ActionScript needed for creating the sample application is provided with this tutorial. However, to understand the scripting concepts and create your own application using the procedures described here, you should have some prior experience writing ActionScript.

View the application

In this tutorial you'll create an application for the "Fix Your Mistake" gift service, which helps users select an appropriate gift when they need to make amends with someone.

Keep in mind that the sample application is for demonstration purposes only. It is not possible to check errors or verify data in the sample.

The sample application uses several UI components (including the ComboBox, TextArea, and Button components) to create the application interface. It includes data components to connect to external data sources: the XMLConnector component (to connect to an XML file) and the DataSet component (to filter the data from the XML file and make the data available to UI components). The application also uses the WebService class to connect dynamically to a web service.

View the SWF file for the completed application

To view the completed application, open the first_app_v3.swf file at the following location:

- In Windows: *boot drive*\Program Files\Macromedia\Flash MX 2004\Samples\HelpExamples\components_application
- On the Macintosh: *Macintosh HD*/Applications/Macromedia Flash MX 2004/Samples/HelpExamples/components_application

To see how the application works, first click the arrow control in the What Did You Do? section. Select from a list of blunders you might have committed (ranging in severity from Forgot to Water Your Plants to Burned Your House Down). This section uses the ComboBox UI component, populated by a web service.

A list of gift suggestions appears in the Gift Ideas section. Click a gift to view more information about it. In the pop-up window that appears, select the quantity you want using the numeric stepper, and click Add to Cart. Click the close box to close the window. Back in the main screen of the application, click Checkout. This section uses the XMLConnector data component to connect to an external XML file, the DataSet data component to filter the data from the XML file, and the DataGrid UI component to display the data.

On the Checkout screen, click the Billing Information, Shipping Information, and Credit Card Information headers to view the form fields for each of these items. To place an order, you can add the appropriate information in each of these panes, and click Confirm at the bottom of the Credit Card Information pane. You can also click Back to return to the main screen. Close the SWF file when you finish examining the completed application. This screen includes several UI components (Accordion, TextArea, and others) to display information and provide fields for user input.

View the FLA file for the completed application

To view the FLA file for the application, open the first_app_v3.fla file in the components_application folder (the same folder that contains the first_app_v3.swf file).

Examine the Stage, library, and Actions panel to see the content for the application. Notice that all the components used in the application appear in the library (along with graphics files and other assets used to create the application architecture). Drag the playhead to view the keyframes labeled Home (Frame 1) and Checkout (Frame 10). Some components appear as instances on the Stage. Some are referenced in the ActionScript code but do not appear until runtime.

About data integration in the sample application

The sample application uses features of the Flash data integration architecture to connect to external data sources, manage the data from those sources, and map the data to UI components in the application for display. The Flash data integration architecture enables you to work with external data in different ways, using components and ActionScript classes. For general information on Flash data integration features, see Chapter 14, "Data Integration (Flash Professional Only)" in *Using Flash*.

The sample application uses both components and classes, to introduce you to different ways of working with data:

- The XMLConnector component connects to an external XML file. Using this component is similar to loading an external XML file with `XML.load()` (the `load` method of the XML object). However, the XMLConnector component is far more powerful and versatile, because the component makes the XML data available for display in a variety of UI components, simply by binding component parameters in the Flash authoring environment. For more information, see "XMLConnector component (Flash Professional only)" on page 894.
- The DataSet component manages and filters data from the XML file. You bind the XMLConnector component to the DataSet component in the Flash authoring environment, and then bind the DataSet component to a UI component. For more information, see "DataSet component (Flash Professional only)" on page 301.
- The DataGrid component displays data from the XML file that has been filtered by the DataSet. You bind the DataSet component to the DataGrid component. (You can also bind the DataSet component to other UI components. The DataGrid component is just one example.) For more information, see "DataGrid component (Flash Professional only)" on page 247.

- The WebService class is part of a set of web service classes, which provides a set of methods, events, and properties that enable you to connect to a web service. The WebService class is different from the WebServiceConnector component. (The WebServiceConnector component, like the XMLConnector component, enables you to connect to an external data source—in this case, a web service—by adding a component to an application and setting its parameters.) The sample application uses the WebService class rather than the WebServiceConnector component simply to demonstrate another way of connecting to an external data source. For more information on the set of web service classes, see "Web service classes (Flash Professional only)" on page 842.

Build the application architecture

To build the application architecture, you'll add components to the Stage on Frame 1 (for the main screen) and Frame 10 (for the Checkout screen). You'll also create movie clips that will be used to display information inside various components.

Add component instances for the main screen of the application

You'll start the application by adding instances of the ComboBox, DataGrid, DataSet, XMLConnector, and Button components to the Stage.

You'll also add the Window component to the library. Later in the tutorial you'll add code to create instances of the Window component dynamically, to display product information when a user clicks an item in the Gift Ideas section.

The ComboBox instance will display the list of blunders that the user can choose from. The list will be provided by a web service that you'll connect to the ComboBox component later in the tutorial, using the WebService class.

The DataGrid instance will display the list of gift ideas that the user can choose from. The list of gifts (and all the product details for each gift) will be provided by an external XML file, which you connect to by means of the XMLConnector component. To filter and sort the data from the XML file, you'll use the DataSet component. Later in the tutorial, you'll use the Flash data binding features to bind the DataGrid, XMLConnector, and DataSet components to interpret and display product information from the XML file.

The Window component will be used to create a pop-up window that displays information on each product in the Gift Ideas list.

1. Open the first_app_v3_start.fla file for the application, located in the components_application folder (the same folder that contains the first_app_v3.swf and first_app_v3.fla files).

   The start_app.fla file contains three layers: a Background layer with a black background image and text titles, a Text layer with text labels for sections of the application, and a Labels layer with labels on the first frame (Home) and the tenth frame (Checkout).
2. Select File > Save As. Rename the file and save it to your hard drive.
3. In the Timeline, select the Labels layer and click the Add Layer button to add a new layer above it. Name the new layer **Form**. You'll place the component instances on this layer.

4. Make sure the Form layer is selected. In the Components panel (Window > Development Panels > Components), locate the ComboBox component in the UI Components folder. Drag an instance of ComboBox onto the Stage. Place it below the What Did You Do? text. In the Property inspector (Window > Properties), for the instance name enter **problems_cb**. Set Width to 400 pixels.

   Notice that the ComboBox component symbol is added to the library (Window > Library). When you drag an instance of a component to the Stage, the compiled clip symbol for the component is added to the library. As with all symbols in Flash, you can create additional instances of the component by dragging the library symbol onto the Stage.

5. Drag an instance of the DataGrid component from the UI Components folder in the Components panel onto the Stage. Place it below the Gift Ideas text. Enter **products_dg** for the instance name. Set Width to 400 pixels and Height to 160 pixels.

6. Drag an instance of the DataSet component from the Data Components folder in the Components panel onto the side of the Stage. (The DataSet component does not appear in the application at runtime. The DataSet icon is simply a placeholder that you work with in the Flash authoring environment.) Enter **products_ds** for the instance name.

7. Drag an instance of the XMLConnector component from the Data Components folder in the Components panel to the side of the Stage. (Like the DataSet component, the XMLConnector component does not appear in the application at runtime.) Enter **products_xmlcon** for the instance name. Click the Parameters tab in the Property inspector, and enter **http://www.flash-mx.com/mm/firstapp/products.xml** for the `URL` property.

   Note: You can also use the Component inspector (Window › Development Panels › Component Inspector) to set parameters for components. The Parameters tab in the Property inspector and the Component inspector work in the same way.

   The URL specifies an external XML file with data about the products that appear in the Gift Ideas section of the application. Later in the tutorial you'll use data binding to bind the XMLConnector, DataSet, and DataGrid components together; the DataSet component will filter data from the external XML file, and the DataGrid component will display it.

8. Drag an instance of the Button component from the UI Components folder in the Components panel onto the Stage. Place it in the lower right corner of the Stage. Enter **checkout_button** for the instance name. Click the Parameters tab and enter `Checkout` for the `label` property.

9. Drag an instance of the Window component from the UI Components folder in the Components panel onto the Stage. Select the instance on the Stage and delete it.

   The Window component symbol is now added to the library. Later in the tutorial, you'll create instances of the Window component using ActionScript.

   Remember to save your work frequently.

Create a movie clip with component instances to display product details

In the application, a pop-up window appears when a user clicks on a product in the Gift Ideas section. The pop-up window contains component instances that display information for the product, including a text description, image, and price. To make this pop-up window, you'll create a movie clip symbol and add instances of the Loader, TextArea, Label, NumericStepper, and Button components.

Later in the tutorial, you'll add ActionScript that dynamically creates an instance of this movie clip for each product. These movie clip instances will be displayed in the Window component, which you added to the library earlier. The component instances will be populated with elements from the external XML file.

1. In the Library panel (Window > Library), click the options menu on the right side of the title bar and select New Symbol.
2. In the Create New Symbol dialog box, enter **ProductForm** for Name and select Movie Clip for Behavior.
3. Click the Advanced button. Under Linkage, select Export for ActionScript, leave Export in First Frame selected, and click OK. A document window for the new symbol opens in symbol-editing mode.

   For movie clip symbols that are in the library but not on the Stage, you must select Export for ActionScript so that you can manipulate them using ActionScript. (Exporting in first frame means that the movie clip is available as soon as the first frame loads.) Later in the tutorial you'll add ActionScript that will generate an instance of the movie clip dynamically each time a user clicks a product in the Gift Ideas section.
4. In the Timeline for the new symbol, select Layer 1 and rename it **Components**.
5. Drag an instance of the Loader component from the UI Components folder in the Components panel onto the Stage. Set the X, Y coordinates to 5, 5. Enter **image_ldr** for the instance name. Click the Parameters tab in the Property inspector. Select `false` for `autoLoad` and `false` for `scaleContent`.

   The Loader component instance will be used to display an image of the product. The `false` setting for `autoLoad` specifies that the image will not load automatically. The `false` setting for `scaleContent` specifies that the image will not be scaled. Later in the tutorial you'll add code that loads the image dynamically, based on the product that the user selects in the Gift Ideas section.
6. Drag an instance of the TextArea component from the UI Components folder in the Components panel onto the Stage. Place it next to the Loader component. Set the X, Y coordinates to 125, 5. Enter **description_ta** for the instance name. Click the Parameters tab in the Property inspector. For `editable`, select `false`. For `html`, select `true`. For `wordWrap`, select `true`.

   The TextArea component instance will be used to display a text description of the selected product. The selected settings specify that the text cannot be edited by a user, that it can be formatted with HMTL tags, and that lines will wrap to fit the size of the text area.

7. Drag an instance of the Label component from the UI Components folder in the Components panel onto the Stage. Place it below the Loader component. Set the X, Y coordinates to 5, 145. Enter **price_lbl** for the instance name. Click the Parameters tab in the Property inspector. For `autoSize`, select `left`. For `html`, select `true`.

   The Label component instance will display the price of the product and the price qualifier (the quantity of products indicated by the specified price, such as "each" or "one dozen.")

8. Drag an instance of the NumericStepper component from the UI Components folder in the Components panel onto the Stage. Place it below the TextArea component. Set the X, Y coordinates to 135, 145. Enter **quantity_ns** for the instance name. Click the Parameters tab in the Property inspector. For `minimum`, enter 1.

   Setting `minimum` to 1 specifies that the user must select at least one of the product in order to add the item to the cart.

9. Drag an instance of the Button component from the UI Components folder in the Components panel onto the Stage. Place it beside the NumericStepper component. Set the X, Y coordinates to 225, 145. Enter **addToCart_button** for the instance name. Click the Parameters tab in the Property inspector. For `label`, enter `Add To Cart`.

Add code to the ProductForm movie clip

Next, you'll add ActionScript to the ProductForm movie clip that you just created. The ActionScript populates the components in the movie clip with information about the selected product, and adds an event listener to the Add to Cart button that adds the selected product to the cart.

For more information on working with event listeners, see "Using event listeners" in *Using ActionScript in Flash*.

1. In the Timeline of the ProductForm movie clip, create a new layer and name it **Actions**. Select the first frame in the Actions layer.
2. In the Actions panel, add the following code:

```
// create an object to reference the selected product item in the DataGrid
var thisProduct:Object = this._parent._parent.products_dg.selectedItem;
// populate the description_ta TextArea and price_lbl Label instances with
// data from the selected product
description_ta.text = thisProduct.description;
price_lbl.text = "<b>$"+thisProduct.price+" "+thisProduct.priceQualifier+"</
  b>";
// load an image of the product from the application directory
image_ldr.load(thisProduct.image);
```

   Note: The code includes comments explaining its purpose. It's a good idea to include comments like these in all the ActionScript code you write, so that you or anyone else going back to the code later can easily understand what it was for.

   First, the code defines a variable to refer to the selected product in the subsequent code. Using `thisProduct` means you don't have to refer to the specified product using the path `this._parent._parent.products_dg.selectedItem`.

Next, the code populates the TextArea and Label instances by using the `description`, `price`, and `priceQualifier` properties of the `thisProduct` object. These properties correspond to elements in the products.xml file that you linked to the products_xmlcon XMLConnector instance at the beginning of the tutorial. Later in the tutorial, you'll bind the XMLConnector, DataSet, and DataGrid component instances together, and the elements in the XML file will populate the other two component instances.

Finally, the code uses the `image` property of the `thisProduct` object to load an image of the product into the Loader component.

3. Next you'll add an event listener to add the product to the cart when the user clicks the Add to Cart button. (You'll add ActionScript to the main Timeline in the application later in the tutorial, to create an instance of the Cart class.) Add the following code:

```
var cartListener:Object = new Object();
cartListener.click = function(evt:Object) {
  var tempObj:Object = new Object();
  tempObj.quantity = evt.target._parent.quantity_ns.value;
  tempObj.id = thisProduct.id;
  tempObj.productObj = thisProduct;
  var theCart = evt.target._parent._parent._parent.myCart;
  theCart.addProduct(tempObj.quantity, thisProduct);
};
addToCart_button.addEventListener("click", cartListener);
```

4. Click the Check Syntax button (the blue check mark above the Script pane) to make sure there are no syntax errors in the code.

   You should check syntax frequently as you add code to an application. Any errors found in the code are listed in the Output panel. (When you check syntax, only the current script is checked; other scripts that may be in the FLA file are not checked.) For more information, see "Debugging your scripts" in *Using ActionScript in Flash*.

5. Click the arrow button at the top left of the document window or select View > Edit Document to exit symbol editing mode and return to the main Timeline.

Add components for the Checkout screen

When the user clicks the Checkout button on the main screen, the Checkout screen appears. The Checkout screen provides forms where the user can enter billing, shipping, and credit card information.

The checkout interface consists of components placed on a keyframe at Frame 10 in the application. You'll use the Accordion component to create the checkout interface. The Accordion component is a navigator that contains a sequence of children that it displays one at a time. You'll also add a Button component instance to create a Back button, so users can return to the main screen.

Later in the tutorial, you'll create movie clips to use as children in the Accordion instance, to display the Billing, Shipping, and Credit Card Information panes.

1. In the main Timeline for the application, move the playhead to Frame 10 (labeled Checkout). Make sure the Form layer is selected.
2. Insert a blank keyframe on Frame 10 in the Form layer (select the frame and select Insert > Timeline > Blank Keyframe).
3. With the new keyframe selected, drag an instance of the Accordion component from the UI Components folder in the Components panel onto the Stage. In the Property inspector, enter **checkout_acc** for the instance name. Set Width to 300 pixels and Height to 200 pixels.
4. Drag an instance of the Button component from the UI Components folder in the Components panel onto the Stage. In the Property inspector, enter **back_button** for the instance name. Click the Parameters tab, and enter `Back` for the `label` property.

About the Billing, Shipping, and Credit Card panes

The Billing, Shipping, and Credit Card Information panes are built with movie clip instances that are displayed in the Accordion component instance. Each pane consists of two nested movie clips.

The parent movie clip contains a ScrollPane component, used to display content in a scrollable area. The child movie clip contains Label and TextInput components where users can enter personal data, such as name, address, and so on. You'll use the ScrollPane component to display the child movie clip so that the user can scroll through the information fields.

Create movie clips for the Billing Information pane

First you'll create two movie clips that will display the Billing Information form fields: a parent movie clip with the ScrollPane component instance, and a child movie clip with the Label and TextArea component instances.

1. In the Library panel (Window > Library), click the options menu on the right side of the title bar and select New Symbol.
2. In the Create New Symbol dialog box, enter **checkout1_mc** for Name and select Movie Clip for Behavior.
3. Click the Advanced button. Under Linkage, select Export for ActionScript, leave Export in First Frame selected, and click OK.

   A document window for the new symbol opens in symbol-editing mode.
4. Drag an instance of the ScrollPane component onto the Stage.
5. In the Property inspector, enter **checkout1_sp** for the instance name. Set the W and H coordinates to 300, 135. Set the X and Y coordinates to 0, 0.
6. Click the Parameters tab. Set the `contentPath` property to `checkout1_sub_mc`.

   The checkout1_sub_mc movie clip will appear inside the scroll pane, and will contain the Label and TextInput components. You'll create this movie clip next.
7. From the Library options menu, select New Symbol.

8. In the Create New Symbol dialog box, enter **checkout1_sub_mc** for Name and select Movie Clip for Behavior.
9. Click the Advanced button. Under Linkage, select Export for ActionScript, leave Export in First Frame selected, and click OK.

   A document window for the new symbol opens in symbol-editing mode.
10. Drag six instances of the Label component onto the Stage. Alternatively, you can drag one instance onto the Stage, and Control-drag (Windows) or Option-drag (Macintosh) it on the Stage to make copies. Name and position the instances as follows:
    - For the first instance, enter **firstname_lbl** for the instance name and set the X and Y coordinates to 5, 5. Click the Parameters tab and enter `First Name` for `text`.
    - For the second instance, enter **lastname_lbl** for the instance name and set the X and Y coordinates to 5, 35. Click the Parameters tab and enter `Last Name` for `text`.
    - For the third instance, enter **country_lbl** for the instance name and set the X and Y coordinates to 5, 65. Click the Parameters tab and enter `Country` for `text`.
    - For the fourth instance, enter **province_lbl** for the instance name and set the X and Y coordinates to 5, 95. Click the Parameters tab and enter `Province/State` for `text`.
    - For the fifth instance, enter **city_lbl** for the instance name and set the X and Y coordinates to 5, 125. Click the Parameters tab and enter `City` for `text`.
    - For the sixth instance, enter **postal_lbl** for the instance name and set the X and Y coordinates to 5, 155. Click the Parameters tab and enter `Postal/Zip Code` for `text`.
11. Drag six instances of the TextInput component onto the Stage. Place a TextInput instance immediately to the right of each Label instance. For example, the X, Y coordinates of the first TextInput instance should be 105, 5. Name the TextInput instances as follows:
    - Name the first instance **billingFirstName_ti**.
    - Name the second instance **billingLastName_ti**.
    - Name the third instance **billingCountry_ti**.
    - Name the fourth instance **billingProvince_ti**.
    - Name the fifth instance **billingCity_ti**.
    - Name the sixth instance **billingPostal_ti**.

    Sometimes content in a scroll pane can be cropped if it's too close to the border of the pane. In the next few steps you'll add a white rectangle to the checkout1_sub_mc movie clip so that the Label and TextInput instances are displayed properly.
12. In the Timeline, click the Add New Layer button. Drag the new layer below the existing layer. (The layer with the rectangle should be on the bottom, so that the rectangle doesn't interfere with the component display.)
13. Select Frame 1 of the new layer.

14. In the Tools panel, select the Rectangle tool. Set the Stroke color to None and the Fill color to white.

    Click the Stroke Color control in the Tools panel and click the None button—the white swatch with a red line through it. Click the Fill Color control and click the white color swatch.

15. Drag to create a rectangle that extends beyond the bottom and right edges of the Label and TextInput instances.

Create movie clips for the Shipping Information pane

The movie clips for the Shipping Information pane are very similar to those for the Billing Information pane. You'll also add a CheckBox component, enabling users to populate the Shipping Information form fields with the same data they entered in the Billing Information pane.

1. Follow the earlier instructions (see "Create movie clips for the Billing Information pane" on page 28) to create the movie clips for the Credit Card Information pane. Note these naming differences:
   - For the first movie clip, enter **checkout2_mc** for the symbol name and **checkout2_sp** for the instance name. In the Property inspector's Parameters tab, set the `contentPath` property to `checkout2_sub_mc`.
   - For the second movie clip, enter **checkout2_sub_mc** for the symbol name.
   - For the TextInput instances, change "billing" to "shipping" in the instance names.
2. With the checkout2_sub_mc movie clip open in symbol-editing mode, drag an instance of the CheckBox component onto the Stage and position it just above the first Label instance.

   Make sure to place this instance on Layer 1, along with the other component instances.
3. In the Property inspector, enter **sameAsBilling_ch** for the instance name.
4. Click the Parameters tab. Set the `label` property to `Same As Billing Info`.

Create movie clips for the Credit Card Information pane

The movie clips for the Credit Card Information pane are also similar to those for the Billing and Shipping Information panes. However, the nested movie clip for the Credit Card Information pane has somewhat different fields than the other two panes, for credit card number and other card data.

1. Follow steps 1-11 of the Billing Information instructions (see "Create movie clips for the Billing Information pane" on page 28) to create the movie clips for the Credit Card Information pane. Note these naming differences:
   - For the first movie clip, enter **checkout3_mc** for the symbol name and **checkout3_sp** for the instance name. In the Property inspector's Parameters tab, set the `contentPath` property to `checkout3_sub_mc`.
   - For the second movie clip, enter **checkout3_sub_mc** for the symbol name.

2. Drag four instances of the Label component onto the Stage. Name and position the instances as follows:
   - For the first instance, enter **ccName_lbl** for the instance name and set the X and Y coordinates to 5, 5. Click the Parameters tab and enter `Name On Card` for `text`.
   - For the second instance, enter **ccType_lbl** for the instance name and set the X and Y coordinates to 5, 35. Click the Parameters tab and enter `Card Type` for `text`.
   - For the third instance, enter **ccNumber_lbl** for the instance name and set the X and Y coordinates to 5, 65. Click the Parameters tab and enter `Card Number` for `text`.
   - For the fourth instance, enter **ccExp_lbl** for the instance name and set the X and Y coordinates to 5, 95. Click the Parameters tab and enter `Expiration` for `text`.
3. Drag an instance of the TextInput component onto the Stage and position it to the right of the ccName_lbl instance. Name the new instance **ccName_cb**.
4. Drag an instance of the ComboBox component onto the Stage and position it to the right of the ccType_lbl instance. Name the new instance **ccType_cb**.
5. Drag another instance of the TextInput component onto the Stage and position it to the right of the ccNumber_lbl instance. Name the new instance **ccNumber_cb**.
6. Drag two instances of the ComboBox component onto the Stage. Position one to the right of the ccExp_lbl instance, and position the other one to the right of that. Name the first new instance **ccMonth_cb**, and name the second **ccYear_cb**.
7. Drag an instance of the Button component onto the Stage and position it at the bottom of the form, below the ccMonth_cb instance. Name the new instance **checkout_button**. In the Property inspector's Parameters tab, set the `label` property to `Checkout`.
8. Follow the instructions in steps 14-16 of the Billing Information instructions (see "Create movie clips for the Billing Information pane" on page 28) to add a rectangle to the bottom of the form.

Bind components to display product information from an external source

In the beginning of the tutorial, you added instances of the DataGrid, DataSet, and XMLConnector components to the Stage. You set the `URL` property for the XMLConnector (named products_xmlcon) to the location of an XML file containing product information for the Gift Ideas section of the application.

Now you'll use data binding features in the Flash authoring environment to bind the XMLConnector, DataSet, and DataGrid components together to use the XML data in the application. For general information on working with data binding and other features of the Flash data integration architecture, see Chapter 14, "Data Integration (Flash Professional Only)" in *Using Flash*.

When you bind the components, the DataSet component will filter the list of products in the XML file according to the severity of the blunder that the user selects in the What Did You Do? section. The DataGrid component will display the list.

Specify a schema for the XML data source

When you connect to an external XML data source with the XMLConnector component, you need to specify a *schema*—a schematic representation which identifies the structure of the XML document. The schema tells the XMLConnector component how to read the XML data source. The easiest way to specify a schema is to import a copy of the XML file that you're going to connect to, and use that copy as a schema.

1. Launch your web browser and go to www.flash-mx.com/mm/firstapp/problems.xml (the location you set for the XMLConnector URL parameter).
2. Select File > Save As.
3. Save products.xml to the same location as the FLA file that you're working on.
4. Select Frame 1 in the main Timeline.
5. On the Stage, select the products_xmlcon (XMLConnector) instance.
6. In the Component inspector, click the Schema tab. Click the Import button (on the right side of the Schema tab, above the scroll pane). In the Open dialog box, locate the products.xml file that you imported in step 3, and click Open. The schema for the products.xml file appears in the scroll pane of the Schema tab.

Bind the XMLConnector, DataSet, and DataGrid components

You'll use the Binding tab in the Component inspector to bind the XMLConnector, DataSet, and DataGrid component instances to one another.

For information on working with data binding, see "Data binding (Flash Professional only)" in *Using Flash*.

1. With the products_xmlcon (XMLConnector) instance selected on the Stage, click the Bindings tab in the Component inspector.
2. Click the Add Binding button.
3. In the Add Binding dialog box, select the `results.products.product array` item and click OK.
4. In the Bindings tab, click the Bound To item in the Binding Attributes pane (the bottom pane, showing attribute name-value pairs).
5. In the Value column for the Bound To item, click the magnifying glass icon to open the Bound To dialog box.
6. In the Bound To dialog box, select the products_ds (DataSet) instance in the Component Path pane. Select `dataProvider:array` in the Schema Location pane. Click OK.
7. In the Bindings tab, click the Direction item in the Binding Attributes pane. From the pop-up menu in the Value column, select Out.

   This option means that the data will pass from the products_xmlcon instance to the products_ds instance (rather than passing in both directions, or passing from the DataSet instance to the XMLConnector instance).

8. On the Stage, select the products_ds instance. In the Bindings tab of the Component inspector, notice that the component's data provider appears in the Binding List (the top pane of the Bindings tab). In the Binding Attributes pane, the Bound To parameter indicates that the products_ds instance is bound to the products_xmlcom instance, and the binding direction is In.

   In the next few steps you'll bind the DataSet instance to the DataGrid instance so that the data that is filtered by the data set will be displayed in the data grid.
9. With the products_ds instance still selected, click the Add Binding button in the Bindings tab.
10. In the Add Binding dialog box, select the `dataProvider: array` item and click OK.
11. In the Bindings tab, make sure the `dataProvider: array` item is selected in the Binding List.
12. Click the Bound To item in the Binding Attributes pane.
13. In the Value column for the Bound To item, click the magnifying glass icon to open the Bound To dialog box.
14. In the Bound To dialog box, select the products_dg (DataGrid) instance in the Component Path pane. Select `dataProvider:array` in the Schema Location pane. Click OK.

Add ActionScript to the main Timeline

With the application architecture and data binding in place, you're ready to add ActionScript to the main Timeline to complete the application functionality.

Create references to component class names

Each component is associated with a class file that defines its methods and properties. In this section of the tutorial, you'll add ActionScript to create references to the class names for the components used in the application. For some of these components, you have already added instances to the Stage. For others, you'll add ActionScript later in the tutorial to create instances dynamically.

Creating a references to the class name makes it easier to write ActionScript for the component because it enables code completion for component instances, so you can avoid using the fully qualified name. For example, when you create a reference to the class file for the ComboBox component, you can refer to instances of the ComboBox with the syntax `instanceName:ComboBox`, rather than `instanceName:mx.controls.ComboBox`.

You'll need these classes:

WebService class This class populates the ComboBox instance with a list of "problems." For this class, you'll also need to import the WebServiceClasses item from the Classes common library. This item contains compiled clips (SWC files) that you'll need in order to compile and generate the SWF file for your application.

UI Components Controls package This package contains classes for the user interface control components, including ComboBox, DataGrid, Loader, TextInput, Label, NumericStepper, Button, and CheckBox. A package is a directory that contains class files and resides in a designated classpath directory. You can use a wild card to create references to all the classes in a package: for example, the syntax `mx.controls.*` creates references to all classes in the controls package. (When you create a reference to a package with a wild card, the unused classes are dropped from the application when it is compiled, so they don't add any extra size.)

UI Components Containers package This package contains classes for the user interface container components, including Accordion, ScrollPane, and Window. As with the controls package, you can create a reference to this package by using a wild card.

DataGridColumn class This class lets you add columns to the DataGrid instance and control their appearance.

Cart class A custom class provided with this tutorial, the Cart class defines the functioning of the shopping cart that you'll create later. (To examine the code in the Cart class file, open the cart.as file located in the component_application folder with the application FLA and SWF files).

You'll create an Actions layer and add ActionScript code to the first frame of the main Timeline. For all the code that you'll add to the application in the remaining steps of the tutorial, make sure you place it on the Actions layer.

1. To import the WebServiceClasses item from the Classes library, select Window > Other Panels > Common Libraries > Classes.
2. Drag the WebServiceClasses item from the Classes library into the library for the application.

   Importing an item from the Classes library is similar to adding a component to the library: it adds the SWC files for the class to the library. The SWC files need to be in the library in order for you to use the class in an application.
3. In the Timeline, select the Form layer and click the Add New Layer button. Name the new layer **Actions**.
4. With the Actions layer selected, select Frame 1 and press F9 to open the Actions panel.
5. In the Actions panel, enter the following code to create a `stop()` function that prevents the application from looping during playback:

   ```
   stop();
   ```
6. With Frame 1 in the Actions layer still selected, add the following code in the Actions panel to import the classes:

   ```
   // import necessary classes
   import mx.services.WebService;
   import mx.controls.*;
   import mx.containers.*;
   import mx.controls.gridclasses.DataGridColumn;
   // import the custom Cart class
   import Cart;
   ```

Add an instance of the Cart class and initialize it

The next code that you'll add creates an instance of the custom Cart class and then initializes the instance.

- In the Actions panel, add the following code:

```
var myCart:Cart = new Cart(this);
myCart.init();
```

This code uses the `init()` method of the Cart class to add a DataGrid instance to the Stage, define the columns, and position the DataGrid instance on the Stage. It also adds a Button component instance and positions it, and adds an Alert handler for the button. (To see the code for the Cart class `init()` method, open the Cart.as file.)

Set the data type of component instances

Next you'll assign data types to each of the component instances you dragged to the Stage earlier in the tutorial.

ActionScript 2.0 uses strict data typing, which means that you assign the data type when you create a variable. Strict data typing makes code hints available for the variable in the Actions panel.

- In the Actions panel, add the following code to assign data types to the four component instances that you already created.

```
// data type instances on the Stage; other instances might be added at
// runtime from the Cart class
var problems_cb:ComboBox;
var products_dg:DataGrid;
var cart_dg:DataGrid;
var products_xmlcon:mx.data.components.XMLConnector;
```

Use styles to customize component appearance

Each component has style properties and methods that let you customize its appearance, including highlight color, font, and font size. You can set styles for individual component instances, or set styles globally to apply to all component instances in an application. For this tutorial you'll set styles globally.

- Add the following code to set styles:

```
// define global styles and easing equations for the problems_cb ComboBox
_global.style.setStyle("themeColor", "haloBlue");
_global.style.setStyle("fontFamily", "Verdana");
_global.style.setStyle("fontSize", 10);
_global.style.setStyle("openEasing", mx.transitions.easing.Bounce.easeOut);
```

This code sets the theme color (the highlight color on a selected item), font, and font size for the components, and also sets the easing for the ComboBox—the way that the drop-down menu appears and disappears when you click the ComboBox title bar.

Add columns to the Gift Ideas section

Now you're ready to add columns to the data grid in the Gift Ideas section of the application, for displaying product information and price.

- In the Actions panel, add the following code to create, configure, and add a Name column and a Price column to the DataGrid instance:

```
// define data grid columns and their default widths in the products_dg
// DataGrid instance
var name_dgc:DataGridColumn = new DataGridColumn("name");
name_dgc.headerText = "Name";
name_dgc.width = 280;

// add the column to the DataGrid
products_dg.addColumn(name_dgc);
var price_dgc:DataGridColumn = new DataGridColumn("price");
price_dgc.headerText = "Price";
price_dgc.width = 100;

// define the function that will be used to set the column's label
// at runtime
price_dgc.labelFunction = function(item:Object) {
  if (item != undefined) {
    return "$"+item.price+" "+item.priceQualifier;
  }
};
products_dg.addColumn(price_dgc);
```

Connect to a web service to populate the combo box

In this section you'll add code to connect to a web service that contains the list of blunders (Forgot to Water Your Plants, and so on). The web service description language (WSDL) file is located at www.flash-mx.com/mm/firstapp/problems.cfc?WSDL. To see how the WSDL is structured, point your browser to the WSDL location.

The ActionScript code passes the web service results to the ComboBox instance for display. A function sorts the blunders in order of severity. If no result is returned from the web service (for example, if the service is down, or the function isn't found), an error message appears in the Output panel.

- In the Actions panel, add the following code:

```
/* Define the web service used to retrieve an array of problems.
This service will be bound to the problems_cb ComboBox instance. */
var problemService:WebService = new WebService("http://www.flash-mx.com/mm/
  firstapp/problems.cfc?WSDL");
var myProblems:Object = problemService.getProblems();

/* If you get a result from the web service, set the field that will be used
  for the column label.
Set the data provider to the results returned from the web service. */
myProblems.onResult = function(wsdlResults:Array) {
  problems_cb.labelField = "name";
  problems_cb.dataProvider = wsdlResults.sortOn("severity", Array.NUMERIC);
```

```
};

// If you are unable to connect to the remote web service, display the
// error messages in the Output panel.
myProblems.onFault = function(error:Object) {
  trace("error:");
  for (var prop in error) {
    trace("  "+prop+" -> "+error[prop]);
  }
};
```

Load the external XML file listing product information

Next you'll add a line of code that causes the XMLConnector instance to load, parse, and bind the contents of the remote products.xml file. This file is located at the URL you entered for the `URL` property of the XMLConnector instance that you created earlier. The file contains information on the products that will appear in the Gift Ideas section of the application.

- Add the following code in the Actions panel:

```
products_xmlcon.trigger();
```

Add an event listener to filter the products displayed in the Gift Ideas section

You'll add an event listener to detect when a user selects a blunder in the What Did You Do? section (the problems_cb ComboBox instance). The listener includes a function that filters the Gift Ideas list according to the blunder the user chooses. Selecting a minor blunder displays a list of modest gifts (such as a CD or flowers); selecting a more serious blunder displays more opulent gifts.

For more information on working with event listeners, see "Using event listeners" in *Using ActionScript in Flash*.

- In the Actions panel, add the following code:

```
/* Define a listener for the problems_cb ComboBox instance.
This listener will filter the products in the DataSet (and DataGrid).
Filtering is based on the severity of the currently selected item in the
  ComboBox. */
var cbListener:Object = new Object();
cbListener.change = function(evt:Object) {
  products_ds.filtered = false;
  products_ds.filtered = true;
  products_ds.filterFunc = function(item:Object) {
    // If the current item's severity is greater than or equal to the
  selected
    // item in the ComboBox, return true.
    return (item.severity>=evt.target.selectedItem.severity);
  };
};

// Add the listener to the ComboBox.
problems_cb.addEventListener("change", cbListener);
```

Resetting the `filtered` property (setting it to `false` and then to `true`) at the beginning of the `change()` function ensures that the function will work properly if the user changes the What Did You Do? selection repeatedly.

The `filterFunc` function checks whether a given item in the array of gifts falls within the severity the user selected in the combo box. If the gift is within the selected severity range, it is displayed in the DataGrid instance (which is bound to the DataSet instance).

The last line of code registers the listener to the problems_cb ComboBox instance.

Add an event listener to display product details

Next you'll add an event listener to the products_dg DataGrid instance to display information about each product. When the user clicks a product in the Gift Ideas section, a pop-up window appears with information about the product.

- In the Actions panel, add the following code:

```
// create a listener for the DataGrid to detect when the row in the
// DataGrid is changed
var dgListener:Object = new Object();
dgListener.change = function(evt:Object) {
  // when the current row changes in the DataGrid, launch a new pop-up
  // window displaying the product's details
  myWindow = mx.managers.PopUpManager.createPopUp(_root,
  mx.containers.Window, true, {title:evt.target.selectedItem.name,
  contentPath:"ProductForm", closeButton:true});
  // set the dimensions of the pop-up window
  myWindow.setSize(340, 210);
  // define a listener that closes the pop-up window when the user clicks
  // the close button
  var closeListener:Object = new Object();
  closeListener.click = function(evt) {
    evt.target.deletePopUp();
  };
  myWindow.addEventListener("click", closeListener);
};
products_dg.addEventListener("change", dgListener);
```

This code creates a new event listener called dgListener, and creates instances of the Window component you added to the library earlier. The title for the new window is set to the product's name. The content path for the window is set to the ProductForm movie clip. The size of the window is set to 340 x 210 pixels.

The code also adds a close button to enable the user to close the window after viewing the information.

Add an event listener to the Checkout button

Now you'll add code to display the Checkout screen when the user clicks the Checkout button.

- In the Actions panel, add the following code:

```
// when the Checkout button is clicked, go to the "checkout" frame label
var checkoutBtnListener:Object = new Object();
checkoutBtnListener.click = function(evt:Object) {
  evt.target._parent.gotoAndStop("checkout");
};
checkout_button.addEventListener("click", checkoutBtnListener);
```

This code specifies that, when the user clicks the Checkout button, the playhead moves to the Checkout label in the Timeline.

Add code for the Checkout screen

Now you're ready to add code to the Checkout screen of the application, on Frame 10 on the main Timeline. This code processes the data that users enter in the Billing, Shipping, and Credit Card Information panes that you created earlier with the Accordion component and other components.

1. In the Timeline, select Frame 10 in the Actions layer and insert a blank keyframe (select Insert > Timeline > Blank Keyframe)
2. Open the Actions panel (F9).
3. In the Actions panel, add the following code:

```
stop();
import mx.containers.*;

// define the Accordion component on the Stage
var checkout_acc:Accordion;
```

4. Next you'll add the first child to the Accordion component instance, to accept billing information from the user. Add the following code:

```
// define the children for the Accordion component
var child1 = checkout_acc.createChild("checkout1_mc", "child1_mc",
  {label:"1. Billing Information"});
var thisChild1 = child1.checkout1_sp.spContentHolder;
```

   The first line calls the `createChild()` method of the Accordion component and creates an instance of the `checkout1_mc` movie clip symbol (which you created earlier) with the instance name `child1_mc` and the label "1. Billing Information". The second line of code creates a shortcut to an embedded ScrollPane component instance.

5. Create the second child for the Accordion instance, to accept shipping information:

```
/* Add the second child to the Accordion.
Add an event listener for the sameAsBilling_ch CheckBox.
This copies the form values from the first child into the second child. */
var child2 = checkout_acc.createChild("checkout2_mc", "child2_mc",
  {label:"2. Shipping Information"});
var thisChild2 = child2.checkout2_sp.spContentHolder;
var checkboxListener:Object = new Object();
```

```
checkboxListener.click = function(evt:Object) {
  if (evt.target.selected) {
    thisChild2.shippingFirstName_ti.text =
  thisChild1.billingFirstName_ti.text;
    thisChild2.shippingLastName_ti.text =
  thisChild1.billingLastName_ti.text;
    thisChild2.shippingCountry_ti.text = thisChild1.billingCountry_ti.text;
    thisChild2.shippingProvince_ti.text =
  thisChild1.billingProvince_ti.text;
    thisChild2.shippingCity_ti.text = thisChild1.billingCity_ti.text;
    thisChild2.shippingPostal_ti.text = thisChild1.billingPostal_ti.text;
  }
};
thisChild2.sameAsBilling_ch.addEventListener("click", checkboxListener);
```

The first two lines of code are similar to the code for creating the Billing Information child: you create an instance of the `checkout2_mc` movie clip symbol, with the instance name `child2_mc` and the label "2. Shipping Information". The second line of code creates a shortcut to an embedded ScrollPane component instance.

Beginning with the third line of code, you add an event listener to the CheckBox instance. If the user clicks the check box, the shipping information uses the data the user entered in the Billing Information pane.

6. Next, create a third child for the Accordion instance, for credit card information:

```
// define the third Accordion child
var child3 = checkout_acc.createChild("checkout3_mc", "child3_mc",
  {label:"3. Credit Card Information"});
var thisChild3 = child3.checkout3_sp.spContentHolder;
```

7. Add this code to create ComboBox instances for the credit card month, year, and type, and populate each with a statically defined array:

```
/* set the values in the three ComboBox instances on the Stage:
ccMonth_cb, ccYear_cb and ccType_cb */
thisChild3.ccMonth_cb.labels = ["01", "02", "03", "04", "05", "06", "07",
  "08", "09", "10", "11", "12"];
thisChild3.ccYear_cb.labels = [2004, 2005, 2006, 2007, 2008, 2009, 2010];
thisChild3.ccType_cb.labels = ["VISA", "MasterCard", "American Express",
  "Diners Club"];
```

8. Finally, add the following code to add event listeners to the Checkout button and the Back button. When the user clicks the Checkout button, the listener object copies the form fields from the Billing, Shipping, and Credit Card Information panes into a LoadVars object that is sent to the server. (The LoadVars class lets you send all the variables in an object to a specified URL.) When the user clicks the Back button, the application returns to the main screen.

```
/* Create a listener for the checkout_button Button instance.
This listener sends all the form variables to the server when the user clicks
  the Checkout button. */
var checkoutListener:Object = new Object();
checkoutListener.click = function(evt:Object);
  evt.target.enabled = false;
  /* Create two LoadVars object instances, which send variables to
  and receive results from the remote server. */
```

```
    var response_lv:LoadVars = new LoadVars();
    var checkout_lv:LoadVars = new LoadVars();
    checkout_lv.billingFirstName = thisChild1.billingFirstName_ti.text;
    checkout_lv.billingLastName = thisChild1.billingLastName_ti.text;
    checkout_lv.billingCountry = thisChild1.billingCountry_ti.text;
    checkout_lv.billingProvince = thisChild1.billingProvince_ti.text;
    checkout_lv.billingCity = thisChild1.billingCity_ti.text;
    checkout_lv.billingPostal = thisChild1.billingPostal_ti.text;
    checkout_lv.shippingFirstName = thisChild2.shippingFirstName_ti.text;
    checkout_lv.shippingLastName = thisChild2.shippingLastName_ti.text;
    checkout_lv.shippingCountry = thisChild2.shippingCountry_ti.text;
    checkout_lv.shippingProvince = thisChild2.shippingProvince_ti.text;
    checkout_lv.shippingCity = thisChild2.shippingCity_ti.text;
    checkout_lv.shippingPostal = thisChild2.shippingPostal_ti.text;
    checkout_lv.ccName = thisChild3.ccName_ti.text;
    checkout_lv.ccType = thisChild3.ccType_cb.selectedItem;
    checkout_lv.ccNumber = thisChild3.ccNumber_ti.text;
    checkout_lv.ccMonth = thisChild3.ccMonth_cb.selectedItem;
    checkout_lv.ccYear = thisChild3.ccYear_cb.selectedItem;

    /* Send the variables from the checkout_lv LoadVars to the remote script
    on the server.
    Save the results in the response_lv instance. */
    checkout_lv.sendAndLoad("http://www.flash-mx.com/mm/firstapp/cart.cfm",
    response_lv, "POST");
    response_lv.onLoad = function(success:Boolean) {
      evt.target.enabled = true;
    };
};
thisChild3.checkout_button.addEventListener("click", checkoutListener);
cart_mc._visible = false;
var backListener:Object = new Object();
backListener.click = function(evt:Object) {
  evt.target._parent.gotoAndStop("home");
}
back_button.addEventListener("click", backListener);
```

Test the application

Congratulations! You've finished building the application. Now you're ready to test it. If you find any errors during testing, check your application against the finished sample (first_app_v3.fla) to correct any mistakes.

Try using the procedures demonstrated in this tutorial to build component-based applications of your own.

1. Select Control > Test Movie or press Control+Enter (Windows) or Command+Return (Macintosh).
2. Click through the application to test it: select a blunder, choose a gift, and add checkout information.

CHAPTER 3
Working with Components

In this chapter, you'll use several Macromedia Flash (FLA) files and ActionScript class files to learn how to add components to a document and set their properties. This chapter also explains a few advanced topics such as using code hints, creating custom focus navigation, managing component depth, and upgrading version 1 components to version 2 architecture.

The files used in this chapter are TipCalulator.fla and TipCalculator.swf. The files are installed in the following locations on your hard disk:

- (Windows) Program Files/Flash MX 2004/Samples/HelpExamples/TipCalculator
- (Macintosh) Applications/Flash MX 2004/Samples/HelpExamples/TipCalculator

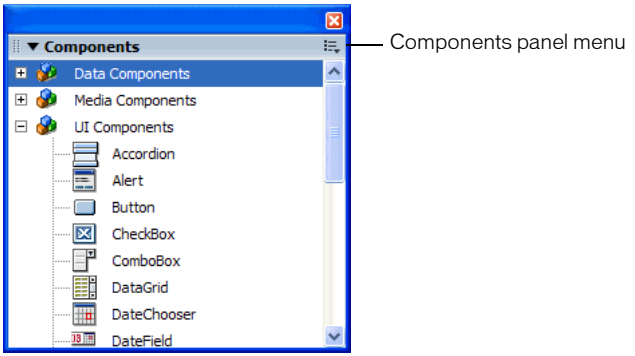
This chapter covers the following topics:

The Components panel

All components in the user-level configuration/Components directory are displayed in the Components panel. (For more information about this directory, see "Where component files are stored" on page 13.)

To display the Components panel:

- Select Window > Development Panels > Components.



To display components that were installed after Flash started up:

1. Select Window > Development Panels > Components.
2. Select Reload from the Components panel menu.

Adding components to Flash documents

When you drag a component from the Components panel to the Stage, a compiled clip (SWC) symbol is added to the Library panel. Once a compiled clip symbol is in the library, you can drag multiple instances to the Stage. You can also add that component to a document at runtime by using the `UIObject.createClassObject()` ActionScript method.

Adding components during authoring

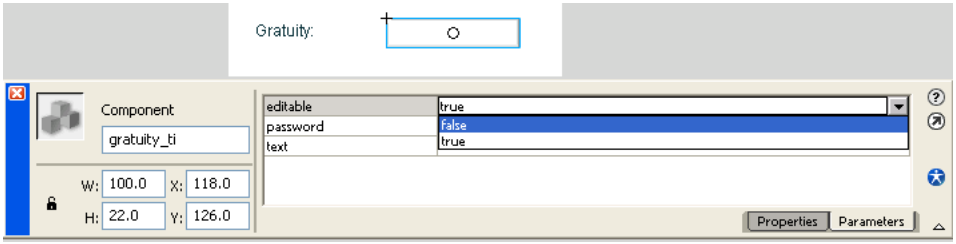
You can add a component to a document by using the Components panel, and then add additional instances of the component to the document by dragging the component from the Library panel to the Stage. You can set properties for additional instances in the Parameters tab of the Property inspector or in the Parameters tab in the Component inspector.

To add a component to a Flash document by using the Components panel:

1. Select Window > Development Panels > Components.
2. Do one of the following:
   - Drag a component from the Components panel to the Stage.
   - Double-click a component in the Components panel.

3. If the component is a FLA (all installed version 2 components are SWC files) *and* if you have edited skins for another instance of the same component, or for a component that shares skins with the component you are adding, do one of the following:
   - Select Don't Replace Existing Items to preserve the edited skins and apply the edited skins to the new component.
   - Select Replace Existing Items to replace all the skins with default skins. The new component and all previous versions of the component, or of components that share its skins, will use the default skins.
4. Select the component on the Stage.
5. Select Window > Properties.
6. In the Property inspector, enter an instance name for the component instance.
7. Click the Parameters tab and specify parameters for the instance.

   The following illustration shows a the Property inspector for the TextInput component that is in the TipCalculator.fla sample file (installed at Flash MX 2004/Samples/HelpExamples/TipCalculator).



   For more information, see "Setting component parameters" on page 47.
8. Change the size of the component as desired.

   For more information on sizing specific component types, see the individual component entries in Chapter 6, "Components Dictionary," on page 91.
9. If you want to change the color and text formatting of a component, do one or more of the following:
   - Set or change a specific style property value for a component instance by using the `setStyle()` method, which is available to all components. For more information, see `UIObject.setStyle()` on page 825.
   - Edit multiple properties in the global style declaration assigned to all version 2 components.
   - Create a custom style declaration for specific component instances.

     For more information, see "Using styles to customize component color and text" on page 67.
10. If you want to customize the appearance of the component, do one of the following:
    - Apply a theme (see "About themes" on page 77).
    - Edit a component's skins (see "About skinning components" on page 80).

Adding components at runtime with ActionScript

Use the `createClassObject()` method (which most components inherit from the UIObject class) to add components to a Flash application dynamically. For example, you could add components that create a page layout based on user-set preferences (as on the home page of a web portal).

Version 2 components that are installed with Flash MX 2004 reside in package directories. (For more information, see "Using packages" in *Using ActionScript in Flash*.) If you add a component to the Stage during authoring, you can refer to the component simply by using its instance name (for example, `myButton`). However, if you add a component to an application with ActionScript (at runtime), you must either specify its fully qualified class name (for example, `mx.controls.Button`) or import the package by using the `import` statement.

For example, to write ActionScript code that refers to an Alert component, you can use the `import` statement to reference the class, as follows:

```
import mx.controls.Alert;
Alert.show("The connection has failed", "Error");
```

Alternatively, you can use the full package path, as follows:

```
mx.controls.Alert.show("The connection has failed", "Error");
```

For more information, see "Importing classes" in *Using ActionScript in Flash*.

You can use ActionScript methods to set additional parameters for dynamically added components. For more information, see Chapter 6, "Components Dictionary," on page 91.

To add a component to a document at runtime, it must be in the library when the SWF file is compiled. To add a component to the library, add it to the Stage and delete it.

Note: The instructions in this section assume an intermediate or advanced knowledge of ActionScript.

To add a component to your Flash document using ActionScript:

1. Drag a component to the Stage and delete it.

   If you do this, you must select the Export in First Frame check box in the Linkage Properties dialog box of the component in the library.

   Note: If a component is set to Export in First Frame, you can't preload the component.

2. Select the frame in the Timeline where you want to add the component.
3. Open the Actions panel if it isn't already open.
4. Call `createClassObject()` to create the component instance at runtime.

   This method can be called on its own, or from any component instance. The `createClassObject()` method takes the following parameters: a component class name, an instance name for the new instance, a depth, and an optional initialization object that you can use to set properties at runtime.

   You can specify the class package in the class name parameter, as in this example:

   ```
   createClassObject(mx.controls.CheckBox, "cb", 5, {label:"Check Me"});
   ```

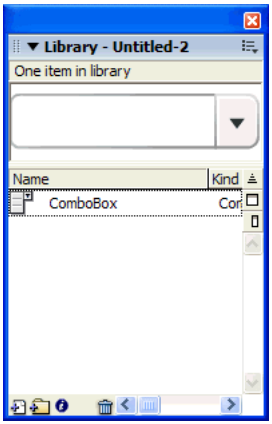
Alternatively, you can import the class package, as in this example:

```
import mx.controls.CheckBox;
createClassObject(CheckBox, "cb", 5, {label:"Check Me"});
```

For more information, see `UIObject.createClassObject()` on page 810 and Chapter 4, "Handling Component Events," on page 55.

Components in the Library panel

When you add a component to a document, it is displayed as a compiled clip (SWC file) symbol in the Library panel.



A ComboBox component in the Library panel

You can add more instances of a component by dragging the component icon from the library to the Stage.

For more information about compiled clips, see "About compiled clips and SWC files" on page 17.

Setting component parameters

Each component has parameters that you can set to change its appearance and behavior. A parameter is a property that appears in the Property inspector and Component inspector. The most commonly used properties appear as authoring parameters; others must be set with ActionScript. All parameters that can be set during authoring can also be set with ActionScript. Setting a parameter with ActionScript overrides any value set during authoring.

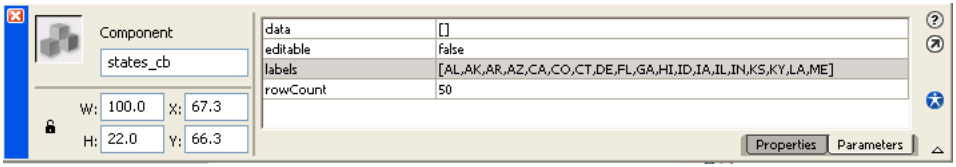
All version 2 components inherit properties and methods from the UIObject and UIComponent classes; these are the properties and methods that all components use, such as `UIObject.setSize()`, `UIObject.setStyle()`, `UIObject.x`, and `UIObject.y`. Each component also has unique properties and methods, some of which are available as authoring parameters. For example, the ProgressBar component has a `percentComplete` property (`ProgressBar.percentComplete`), and the NumericStepper component has `nextValue` and `previousValue` properties (`NumericStepper.nextValue`, `NumericStepper.previousValue`).

You can set parameters for a component instance using the Component inspector or the Property inspector (it doesn't matter which panel you use).

To enter an instance name for a component in the Property inspector:

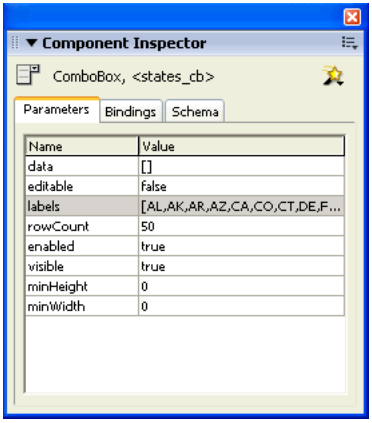
1. Select Window > Properties.
2. Select an instance of a component on the Stage.
3. Enter an instance name in the text field under the word *Component*.

   It's a good idea to add a suffix to the instance name that indicates what kind of component it is; this makes it easier to read your ActionScript code. In this example, the instance name is **states_cb** because the component is a combo box that lists the US states.



To enter parameters for a component instance in the Component inspector:

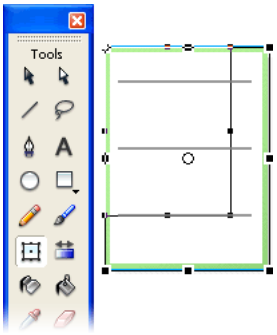
1. Select Window > Development Panels > Component Inspector.
2. Select an instance of a component on the Stage.
3. To enter parameters, click the Parameters tab.



4. To enter or view bindings or schemas for a component, click their respective tabs. For more information, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Sizing components

Use the Free Transform tool or the `setSize()` method to resize component instances.



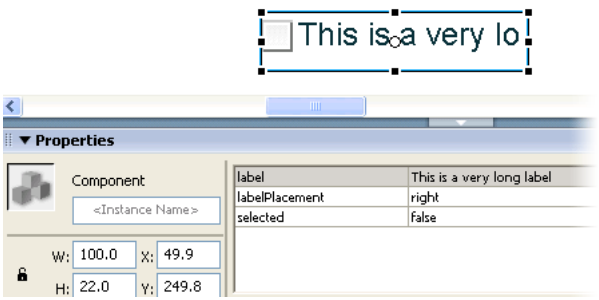
Resizing the Menu component on the Stage with the Free Transform tool

You can call the `setSize()` method from any component instance (see `UIObject.setSize()` on page 823) to resize it. The following code resizes the Menu component to 200 pixels wide and 300 pixels high:

```
myMenu.setSize(200, 300);
```

Note: If you use the ActionScript `_width` and `_height` properties to adjust the width and height of a component, the component is resized but the layout of the content in the component remains the same. This might cause the component to be distorted in movie playback.

A component does not resize automatically to fit its label. If a component instance that has been added to a document is not large enough to display its label, the label text is clipped. You must resize the component to fit its label.



A clipped label for the CheckBox component

For more information about sizing components, see their individual entries in Chapter 6, "Components Dictionary," on page 91.

Deleting components from Flash documents

To delete a component's instances from a Flash document, you must delete the component from the library by deleting the compiled clip icon. It isn't enough to delete the component from the Stage.

To delete a component from a document:

1. In the Library panel, select the compiled clip (SWC) symbol.
2. Click the Delete button at the bottom of the Library panel, or select Delete from the Library options menu.
3. In the Delete dialog box, click Delete to confirm the deletion.

Using code hints

When you are using ActionScript 2.0, you can use strict typing for a variable that is based on a built-in class, including component classes. If you do so, the ActionScript editor displays code hints for the variable. For example, suppose you type the following:

```
import mx.controls.CheckBox;
var myCheckBox:CheckBox;
myCheckBox.
```

As soon as you type the period after `myCheckBox`, Flash displays a list of methods and properties available for CheckBox components, because you have designated the variable as type CheckBox. For more information, see "Strict data typing" and "Using code hints" in *Using ActionScript in Flash*.

Creating custom focus navigation

When a user presses the Tab key to navigate in a Flash application or clicks in an application, the FocusManager class determines which component receives input focus. You don't need to add a FocusManager instance to an application or write any code to activate the Focus Manager.

If a RadioButton object receives focus, the Focus Manager examines that object and all objects with the same `groupName` value and sets focus on the object with the `selected` property set to `true`.

Each modal Window component contains an instance of the Focus Manager, so the controls on that window become their own tab set. This prevents a user from inadvertently navigating to components in other windows by pressing the Tab key.

To create focus navigation in an application, set the `tabIndex` property on any components (including buttons) that should receive focus. When a user presses the Tab key, the FocusManager class looks for an enabled object whose `tabIndex` value is greater than the current value of `tabIndex`. Once the FocusManager class reaches the highest `tabIndex` property, it returns to 0. For example, in the following code, the `comment` object (probably a TextArea component) receives focus first, and then the `okButton` object receives focus:

```
var comment:mx.controls.TextArea;
var okButton:mx.controls.Button;
```

```
comment.tabIndex = 1;
okButton.tabIndex = 2;
```

You can also use the Accessibility panel to assign a tab index value.

If nothing on the Stage has a tab index value, the Focus Manager uses the depth levels (*z*-order). The depth levels are set up primarily by the order in which components are dragged to the Stage; however, you can also use the Modify > Arrange > Bring to Front/Send to Back commands to determine the final *z*-order.

To give focus to a component in an application, call `focusManager.setFocus()`.

To create a button that receives focus when a user presses Enter (Windows) or Return (Macintosh), set the `FocusManager.defaultPushButton` property to the instance of the desired button, as in the following code:

```
focusManager.defaultPushButton = okButton;
```

The FocusManager class overrides the default Flash Player focus rectangle and draws a custom focus rectangle with rounded corners.

For more information about creating a focus scheme in a Flash application, see "FocusManager class" on page 419.

Managing component depth in a document

If you want to position a component in front of or behind another object in an application, you must use the DepthManager class. The methods of the DepthManager class allows you to place user interface components in an appropriate *z*-order (for example, a combo box drops down in front of other components, insertion points appear in front of everything, dialog boxes float over content, and so on).

The Depth Manager has two main purposes: to manage the relative depth assignments within any document, and to manage reserved depths on the root Timeline for system-level services such as the cursor and tooltips.

To use the Depth Manager, call its methods (see "DepthManager class" on page 406).

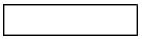
The following code places the component instance `loader` below the `button` component:

```
loader.setDepthBelow(button);
```

Components in Live Preview

The Live Preview feature, enabled by default, lets you view components on the Stage as they will appear in the published Flash content; the components appear at their approximate size. The live preview reflects different parameters for different components. For information about which component parameters are reflected in the live preview, see each component entry in Chapter 6, "Components Dictionary," on page 91.

A Button component with Live Preview enabled



A Button component with Live Preview disabled

Components in Live Preview are not functional. To test component functionality, you can use the Control > Test Movie command.

To turn Live Preview on or off:

- Select Control > Enable Live Preview. A check mark next to the option indicates that it is enabled.

About using a preloader with components

Components are set to Export in First Frame by default. This causes the components to load before the first frame of an application is rendered. If you want to create a preloader for an application, deselect Export in First Frame for any compiled clip symbols in your library.

Note: If you're using the ProgressBar component to display loading progress, leave Export in First Frame selected for the progress bar.

About loading components

If you load version 2 components into a SWF or into the Loader component, the components may not work correctly. These components include the following: Alert, ComboBox, DateField, Menu, MenuBar, and Window.

Use the `_lockroot` property when calling `loadMovie()` or loading into the Loader component. If you're using the Loader component, add the following code:

```
myLoaderComponent.content._lockroot = true;
```

If you're using a movie clip with a call to `loadMovie()`, add the following code:

```
myMovieClip._lockroot = true;
```

If you don't set `_lockroot` to `true` in the loader movie, the loader only has access to its own library, but not the library in the loaded movie.

The `_lockroot` property is supported by Flash Player 7. For information about this property, see `MovieClip._lockroot` in *Flash ActionScript Language Reference*.

Upgrading version 1 components to version 2 architecture

The version 2 components were written to comply with several web standards (regarding events [www.w3.org/TR/DOM-Level-3-Events/events.html], styles, getter/setter policies, and so on) and are very different from their version 1 counterparts that were released with Macromedia Flash MX and in the DRKs that were released before Macromedia Flash MX 2004. Version 2 components have different APIs and were written in ActionScript 2.0. Therefore, using version 1 and version 2 components together in an application can cause unpredictable behavior. For information about upgrading version 1 components to use version 2 event handling, styles, and getter/setter access to the properties instead of methods, see Chapter 7, "Creating Components," on page 915.

Flash applications that contain version 1 components work properly in Flash Player 6 and Flash Player 7, when published for Flash Player 6 or Flash Player 6 (6.0.65.0). If you want to update your applications to work when published for Flash Player 7, you must convert your code to use strict data typing. For more information, see Chapter 10, "Creating Custom Classes with ActionScript 2.0," in *Using ActionScript in Flash.*

CHAPTER 4
Handling Component Events

Every component has events that are broadcast when a user interacts with it (for example, the `click` and `change` events) or when something significant happens to the component (for example, the `load` event). To handle an event, you write ActionScript code that executes when the event occurs.

Each component broadcasts its own set of events. This set includes the events of any class from which the component inherits. This means that all components, except the media components, inherit events from the UIObject and UIComponent classes, because they are the base classes of the version 2 architecture. To see the list of events a component broadcasts, see the component's entry and its ancestor classes' entries in Chapter 6, "Components Dictionary," on page 91.

This chapter uses several versions of a simple Macromedia Flash application, TipCalculator, to teach you how to handle component events. The FLA and SWF files are installed with Flash MX 2004 version 7.2 to the Macromedia/Flash MX 2004/Samples/HelpExamples/TipCalculator folder.

This chapter contains the following sections:

Using the on() event handler

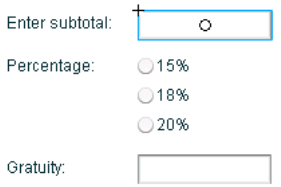
The easiest, but least powerful, way to handle a component event is to use the `on()` event handler. You can assign the `on()` event handler to a component instance, just as you would assign a handler to a button or movie clip. For complex applications, it's best to use event listeners. For more information, see "Using listeners to handle events" on page 56.

The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the Button component instance `myButton`, sends "_level0.myButton" to the Output panel:

```
on(click){
  trace(this);
}
```

To use the on() event handler:

1. Open the file TipCalculator1.fla from Macromedia\Flash MX 2004\Samples\HelpExamples\TipCalculator.
2. On the Stage, select the TextInput component beside the "Enter subtotal" text.



3. Open the Actions panel, if it isn't already open.
4. Look at the following code assigned to the subtotal_ti TextInput component:

   ```
   on(change){
     this._parent.calculate();
   }
   ```

   This code calls the `calculate()` function that is defined on Frame 1 of the main Timeline when the TextInput component changes. The `calculate()` function calculates the tip according to which radio button is selected.
5. Select each of the radio buttons to see their event handlers.

   Each radio button also calls the `calculate()` function when clicked.
6. Select Control > Test Movie to use the tip calculator.

Using listeners to handle events

The version 2 component architecture has a broadcaster/listener event model. (A *broadcaster* is sometimes also referred to as a *dispatcher*.) It is important to understand the following key points about the model:

- All events are broadcast by an instance of a component class. (The component instance is the *broadcaster*.)
- A *listener* can be a function or an object. If the listener is an object, it must have a callback function defined on it. The listener *handles* the event; this means the function, or callback function, executes when the event occurs.

- To register a listener to a broadcaster, call the `addEventListener()` method from the broadcaster. Use the following syntax:

  ```
  componentInstance.addEventListener("eventName", listenerObjectORFunction);
  ```

- You can register multiple listeners to one component instance.

  ```
  myButton.addEventListener("click", listener1);
  myButton.addEventListener("click", listener2);
  ```

- You can register one listener to multiple component instances.

  ```
  myButton.addEventListener("click", listener1);
  myButton2.addEventListener("click", listener1);
  ```

- The handler function is passed an event object.

  You can use the event object in the body of the function to retrieve information about the event type, and the instance that broadcast the event. See "About the event object" on page 66.

Using listener objects

To use a listener object, you can either use the `this` keyword to specify the current object as the listener, use an object that already exists in your application, or create a new object.

- Use `this` in most situations.

  It's often easiest to use the current object (`this`) as a listener, because its scope contains the components that need to react when the event is broadcast.

- Use an existing object if it is convenient.

  For example, in a Flash Form Application, you may want to use a form as a listener object if that form contains the components that react to the event. Place the code on a frame of the form's Timeline.

- Use a new listener object if many components are broadcasting an event (for example, the `click` event) and you want only certain listener objects to respond.

If you use the `this` object, define a function with the same name as the event you want to handle; the syntax is as follows:

```
function eventName(evtObj:Object){
  // your code here
};
```

If you want to use a new listener object, you must create the object, define a property with the same name as the events, and assign the property to a callback function that executes when the event is broadcast, as follows:

```
var listenerObject:Object = new Object();
listenerObject.eventName = function(evtObj:Object){
  // your code here
};
```

If you want to use an existing object, use the same syntax as a new listener object, without creating the new object, as shown here:

```
existingObject.eventName = function(evtObj:Object){
```

```
  // your code here
};
```

Tip: The *evtObj* parameter is an object that is automatically generated when an event is triggered and passed to the callback function. The event object has properties that contain information about the event. For details, see "About the event object" on page 66.

Finally, you call the `addEventListener()` method from the component instance that broadcasts the event. The `addEventListener()` method takes two parameters: a string indicating the name of the event and a reference to the listener object.

```
componentInstance.addEventListener("eventName", listenerObject);
```

Here is the whole code segment, which you can copy and paste. Be sure to replace any code in italics with actual values; you can use `listenerObject` and `evtObj` or any other legal identifiers, but you must change *eventName* to the name of the event.

```
var listenerObject:Object = new Object();
listenerObject.eventName = function(evtObj:Object){
  // code placed here executes
  // when the event is triggered
};
componentInstance.addEventListener("eventName", listenerObject);
```

The following code segment uses the `this` keyword as the listener object:

```
function eventName(evtObj:Object){
  // code placed here executes
  // when the event is triggered
}
componentInstance.addEventListener("eventName", this);
```

You can call `addEventListener()` from any component instance; it is mixed in to every component from the EventDispatcher class. (A "mix-in" is a class that provides specific features that augment the behavior of another class.) For more information, see `EventDispatcher.addEventListener()` on page 416.

For information about the events a component broadcasts, see the component's entry in Chapter 6, "Components Dictionary," on page 91. For example, Button component events are listed in the Button component section (or Help > Using Components > Components Dictionary > Button component > Button class > Event summary for the Button class).

To register a listener object in a Flash (FLA) file:

1. In Flash, select File > New and create a new Flash document.
2. Drag a Button component to the Stage from the Components panel.
3. In the Property inspector, enter the instance name **myButton**.
4. Drag a TextInput component to the Stage from the Components panel.
5. In the Property inspector, enter the instance name **myText**.
6. Select Frame 1 in the Timeline.
7. Select Window > Development Panels > Actions.

8. In the Actions panel, enter the following code:

```
var myButton:mx.controls.Button;
var myText:mx.controls.TextInput;

function click(evt){
  myText.text = evt.target;
}

myButton.addEventListener("click", this);
```

The `target` property of the event object, `evt`, is a reference to the instance broadcasting the event. This code displays the value of the `target` property in the TextInput component.

To register a listener object in a class (AS) file:

1. Open the file TipCalculator.fla from the location specified in "Working with Components" on page 43.
2. Open the file TipCalculator.as from the location specified in "Working with Components" on page 43.
3. In the FLA file, select form1 and view the class name, TipCalculator, in the Property inspector.

   This is the link between the form and the class file. All the code for this application is in the file TipCalculator.as. The form assumes the properties and behaviors defined by the class assigned to it.
4. In the AS file, scroll to line 25, `public function onLoad():Void`.

   The `onLoad()` function executes when the form loads into Flash Player. In the body of the function, the `subtotal` TextInput instance and the three RadioButton instances, `percentRadio15`, `percentRadio18`, and `percentRadio20`, call the `addEventListener()` method to register a listener with an event.
5. Look at line 27, `subtotal.addEventListener("change", this)`.

   When you call `addEventListener()`, you must pass it two parameters. The first is a string indicating the name of the event that is broadcast—in this case, `"change"`. The second is a reference to either an object or a function that handles the event. In this case, the parameter is the keyword `this`, which refers to an instance of the class file (an object). Flash then looks on the object for a function with the name of the event.
6. Look at line 63, `public function change(event:Object):Void`.

   This is the function that executes when the `subtotal` TextInput instance changes.
7. Select the TipCalculator.fla and select Control > Test Movie to test the file.

Using the handleEvent callback function

You can also use listener objects that support a `handleEvent` function. Regardless of the name of the event that is broadcast, the listener object's `handleEvent` method is called. You must use an `if else` or a `switch` statement to handle multiple events. For example, the following code uses an `if else` statement to handle the `click` and `change` events:

```
// define the handleEvent function
// pass it evt as the event object parameter
```

```
function handleEvent(evt){
  // check if the event was a click
  if (evt.type == "click"){
    // do something if the event was click
  } else if (evt.type == "change"){
    // do something else if the event was change
  }
};

// register the listener object to
// two different component instances
// because the function is defined on
// "this" object, the listener is this.

instance.addEventListener("click", this);
instance2.addEventListener("change", this);
```

Using listener functions

Unlike the `handleEvent` syntax, several listener functions can handle different events. So instead of having the `if` and `else if` checks in `myHandler`, you can just define `myChangeHandler` for the `change` event and `myScrollHandler` for the `scroll` event and register them, as shown here:

```
myList.addEventListener("change", myChangeHandler);
myList.addEventListener("scroll", myScrollHandler);
```

To use a listener function, you must first define a function:

```
function myFunction:Function(evtObj:Object){
  // your code here
}
```

Tip: The *`evtObj`* parameter is an object that is automatically generated when an event is triggered and passed to the function. The event object has properties that contain information about the event. For details, see "About the event object" on page 66.

Then you call the `addEventListener()` method from the component instance that broadcasts the event. The `addEventListener()` method takes two parameters: a string indicating the name of the event and a reference to the function.

```
componentInstance.addEventListener("eventName", myFunction);
```

You can call `addEventListener()` from any component instance; it is mixed in to every component from the EventDispatcher class. For more information, see `EventDispatcher.addEventListener()` on page 416.

For information about the events a component broadcasts, see each component's entry in Chapter 6, "Components Dictionary," on page 91.

To register a listener object in a Flash (FLA) file:

1. In Flash, select File > New and create a new Flash document.
2. Drag a List component to the Stage from the Components panel.
3. In the Property inspector, enter the instance name **myList**.

4. Select Frame 1 in the Timeline.
5. Select Window > Development Panels > Actions.
6. In the Actions panel, enter the following code:

```
// declare variables
var myList:mx.controls.List;
var myHandler:Function;

// add items to the list
myList.addItem("Bird");
myList.addItem("Dog");
myList.addItem("Fish");
myList.addItem("Cat");
myList.addItem("Ape");
myList.addItem("Monkey");

// define myHandler function
function myHandler(eventObj:Object){

  // use the eventObj parameter
  // to capture the event type
  if (eventObj.type == "change"){
      trace("The list changed");
  } else if (eventObj.type == "scroll"){
      trace("The list was scrolled");
  }
}

// Register the myHandler function with myList.
// When an item is selected (triggers the change event) or the
// list is scrolled, myHandler executes.
myList.addEventListener("change", myHandler);
myList.addEventListener("scroll", myHandler);
```

Note: The `type` property of the event object, `evt`, is a reference to the event name.

7. Select Control > Test Movie; then select an item in the list and scroll the list to see the results in the Output panel.

Caution: In a listener function, the keyword `this` refers to the component instance that calls `addEventListener()`, not to the Timeline or the class where the function is defined. However, you can use the Delegate class to delegate the listener function to a different scope. See "Delegating events" on page 63. To see an example of function scoping, see the next section.

About scope in listeners

Scope refers to the object within which a function executes. Any variable references within that function are looked up as properties of that object. You can use the Delegate class to specify the scope of a listener. For more information, see "Delegating events" on page 63.

As discussed earlier, you register a listener with a component instance by calling `addEventListener()`. This method takes two parameters: a string indicating the name of the event, and a reference to either an object or a function. The following table lists the scope of each parameter type:

Listener type	Scope
Object	Listener object
Function	Component instance broadcasting the event

If you pass `addEventListener()` an object, the callback function assigned to that object (or the function defined on that object) is invoked in the scope of the object. This means that the keyword `this`, when used inside the callback function, refers to the listener object, as follows:

```
var lo:Object = new Object();
lo.click = function(evt){
  // this refers to the object lo
  trace(this);
}
myButton.addEventListener("click", lo);
```

However, if you pass `addEventListener()` a function, the function is invoked in the scope of the component instance that calls `addEventListener()`. This means that the keyword `this`, when used inside the function, refers to the broadcasting component instance. This causes a problem if you're defining the function in a class file. You cannot access the properties and methods of the class file with the expected paths because `this` doesn't point to an instance of the class. To work around this problem, use the Delegate class to delegate a function to the correct scope. See "Delegating events" on page 63.

The following code illustrates the scoping of a function when passed to `addEventListener()` in a class file. To use this code, copy it into an ActionScript (AS) file named Cart.as. Create a Flash (FLA) file with a Button component, `myButton`, and a DataGrid component, `myGrid`. Select both components on the Stage and press F8 to convert them into a new symbol named `Cart`. In the Linkage properties for the Cart symbol, assign it the class `Cart`.

```
class Cart extends MovieClip {

  var myButton:mx.controls.Button;
  var myGrid:mx.controls.DataGrid;

  function myHandler(eventObj:Object){

  // Use the eventObj parameter
  // to capture the event type.
    if (eventObj.type == "click"){

    /* Send the value of this to the Output panel.
    Because myHandler is a function that is not defined
    on a listener object, this is a reference to the
    component instance to which myHandler is registered
    (myButton). Also, since this doesn't reference an
    instance of the Cart class, myGrid is undefined.
    */
```

```
        trace("this: " + this);
        trace("myGrid: " + myGrid);
      }
    }

    // register the myHandler function with myButton
    // when the button is clicked, myHandler executes

    function onLoad():Void{
      myButton.addEventListener("click", myHandler);
    }
}
```

Delegating events

You can import the Delegate class into your scripts or classes to delegate events to specific scopes and functions. Use the following syntax:

```
import mx.utils.Delegate;
compInstance.addEventListener("eventName", Delegate.create(scopeObject,
  function));
```

The *scopeObject* parameter specifies the scope in which the specified *function* parameter is called.

There are two common uses for calling `Delegate.create()`:

- To dispatch the same event to two different functions.

  See the next section.

- To call functions within the scope of the containing class.

  When you pass a function as a parameter to `addEventListener()`, the function is invoked in the scope of the broadcaster component instance, not the object in which it is declared. See "Delegating the scope of a function" on page 65.

Delegating events to functions

Calling `Delegate.create()` is useful if you have two components that broadcast events of the same name. For example, if you have a check box and a button, you would have to use the `switch` statement on the information you get from the `eventObject.target` property in order to determine which component is broadcasting the `click` event.

To use the following code, place a check box named `myCheckBox_chb` and a button named `myButton_btn` on the Stage. Select both instances and press F8 to create a new symbol. Click Advanced, Export for ActionScript, and enter the AS 2.0 class name **Cart**. You can give the new symbol any instance name you want in the Property inspector. The symbol is now an instance of the Cart class.

```
import mx.utils.Delegate;
import mx.controls.Button;
import mx.controls.CheckBox;

class Cart {
  var myCheckBox_chb:CheckBox;
```

```
  var myButton_btn:Button;

  function onLoad() {
    myCheckBox_chb.addEventListener("click", this);
    myButton_btn.addEventListener("click", this);
  }

  function click(eventObj:Object) {
    switch(eventObj.target) {
      case myButton_btn:
        // sends the broadcaster instance name
        // and the event type to the Output panel
        trace(eventObj.target + ": " + eventObj.type);
        break;
      case myCheckBox_chb:
        trace(eventObj.target + ": " + eventObj.type);
        break;
    }
  }
}
```

The following is the same class file (Cart.as) modified to use Delegate:

```
import mx.utils.Delegate;
import mx.controls.Button;
import mx.controls.CheckBox;

class Cart {
  var myCheckBox_chb:CheckBox;
  var myButton_btn:Button;

  function onLoad() {
    myCheckBox_chb.addEventListener("click", Delegate.create(this,
  chb_onClick));
    myButton_btn.addEventListener("click", Delegate.create(this,
  btn_onClick));
  }

// two separate functions handle the events

  function chb_onClick(eventObj:Object) {
    // sends the broadcaster instance name
    // and the event type to the Output panel
    trace(eventObj.target + ": " + eventObj.type);
    // sends the absolute path of the symbol
    // that you associated with the Cart class
    // in the FLA file to the Output panel
    trace(this)
  }

  function btn_onClick(eventObj:Object) {
    trace(eventObj.target + ": " + eventObj.type);
  }
}
```

Delegating the scope of a function

The `addEventListener()` method requires two parameters: the name of an event and a reference to a listener. The listener can either be an object or a function. If you pass an object, the callback function assigned to the object is invoked in the scope of the object. However, if you pass a function, the function is invoked in the scope of the component instance that calls `addEventListener()`. (For more information, see "About scope in listeners" on page 61.)

Because the function is invoked in the scope of the broadcaster instance, the keyword `this` in the body of the function points to the broadcaster instance, not to the class that contains the function. Therefore, you cannot access the properties and methods of the class that contains the function. Use the Delegate class to delegate the scope of a function to the containing class so that you can access the properties and methods of the containing class.

The following example uses the same approach as the previous section with a variation of the Cart.as class file:

```
import mx.controls.Button;
import mx.controls.CheckBox;

class Cart {

  var myCheckBox_chb:CheckBox;
  var myButton_btn:Button;

  // define a variable to access
  // from the chb_onClick function
  var i:Number = 10

  function onLoad() {
    myCheckBox_chb.addEventListener("click", chb_onClick);
  }

  function chb_onClick(eventObj:Object) {
    // You would expect to be able to access
    // the i variable and output 10.
    // However, this sends undefined
    // to the Output panel because
    // the function isn't scoped to
    // the Cart instance where i is defined.
    trace(i);
  }
}
```

To access the properties and methods of the Cart class, call `Delegate.create()` as the second parameter of `addEventListener()`, as follows:

```
import mx.utils.Delegate;
import mx.controls.Button;
import mx.controls.CheckBox;

class Cart {
  var myCheckBox_chb:CheckBox;
  var myButton_btn:Button;
  // define a variable to access
```

```
  // from the chb_onClick function
  var i:Number = 10

  function onLoad() {
    myCheckBox_chb.addEventListener("click", Delegate.create(this,
  chb_onClick));
  }

  function chb_onClick(eventObj:Object) {
    // Sends 10 to the Output panel
    // because the function is scoped to
    // the Cart instance
    trace(i);
  }
}
```

About the event object

The event object is an instance of the ActionScript Object class; it has the following properties that contain information about an event.

Property	Description
`type`	A string indicating the name of the event.
`target`	A reference to the component instance broadcasting the event.

When an event has additional properties, they are listed in the event's entry in the Components Dictionary.

The event object is automatically generated when an event is triggered and passed to the listener object's callback function or the listener function.

You can use the event object inside the function to access the name of the event that was broadcast, or the instance name of the component that broadcast the event. From the instance name, you can access other component properties. For example, the following code uses the `target` property of the `evtObj` event object to access the `label` property of the `myButton` instance and sends the value to the Output panel:

```
var myButton:mx.controls.Button;
var listener:Object;

listener = new Object();

listener.click = function(evtObj){
  trace("The " + evtObj.target.label + " button was clicked");
}
myButton.addEventListener("click", listener);
```

CHAPTER 5
Customizing Components

You might want to change the appearance of components as you use them in different applications. There are three ways to accomplish this in Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004:

- Use the `setStyle()` method of each component and style declaration to change the color and text formatting of a component. See "Using styles to customize component color and text" on page 67.
- Apply a theme—a collection of styles and skins that make up a component's appearance. See "About themes" on page 77.
- Modify or replace a component's skins. Skins are symbols used to display components. *Skinning* is the process of changing the appearance of a component by modifying or replacing its source graphics. A skin can be a small piece, like a border's edge or corner, or a composite piece like the entire picture of a button in its up state (the state in which it hasn't been pressed). A skin can also be a symbol without a graphic, which contains code that draws a piece of the component. See "About skinning components" on page 80.

Using styles to customize component color and text

Every component instance has style properties (also called *styles*) and `setStyle()` and `getStyle()` methods that you can use to set and get style property values. The `setStyle()` and `getStyle()` methods are inherited from the UIObject class. (For more information, see `UIObject.setStyle()` on page 825 and `UIObject.getStyle()` on page 814.)

Note: You cannot set styles for the Media components.

The styles used by each component depend partially on what theme the document uses (see "About themes" on page 77). Some styles, such as `defaultIcon`, are used by the associated components regardless of the theme applied to the document. Other styles, such as `themeColor` and `symbolBackgroundColor`, are used only by components if the corresponding theme is in use. For example, `themeColor` is used only if the Halo theme is in use, and `symbolBackgroundColor` is used only if the Sample theme is in use.

You can use styles to customize a component in the following ways:

- Set styles on a component instance.

  You can change color and text properties of a single component instance. This is effective in some situations, but it can be time consuming if you need to set individual properties on all the components in a document.

  See "Setting styles on a component instance" on page 69.
- Create custom style declarations and apply them to several component instances.

  You may want to have groups of components in a document share a style. To do this, you can create custom style declarations to apply to the components you specify.

  See "Setting custom styles for groups of components" on page 69.
- Create default class style declarations.

  You can define a default class style declaration so that every instance of a class shares a default appearance.

  See "Setting styles for a component class" on page 71.
- Use inheriting styles to set styles for components in a portion of a document.

  The values of style properties set on containers are inherited by contained components.

  See "Setting inheriting styles on a container" on page 71.
- Use the global style declaration that sets styles for all components in a document.

  If you want to apply a consistent look to an entire document, you can create styles on the global style declaration.

  See "Setting global styles" on page 73.

Flash does not display changes made to style properties when you view components on the Stage using the Live Preview feature. For more information, see "Components in Live Preview" on page 52.

Supported styles

Flash MX 2004 and Flash MX Professional 2004 provide many styles for customizing component color, text, and behavior. The set of styles used depends on each component and the theme applied to the document. For a list of styles supported by each component, see Chapter 6, "Components Dictionary," on page 91.

Flash provides two visual themes for components: Halo (HaloTheme.fla) and Sample (SampleTheme.fla). A *theme* is a set of styles and graphics that controls the appearance of components in a document. Each theme provides additional styles to the components. To know what style properties you can set for a component, you must know what theme is assigned to that component. The style tables for each component in the Components Dictionary indicate whether each style property applies to one or both of the supplied themes. (For more information, see "About themes" on page 77.)

Setting styles on a component instance

You can write ActionScript code to set and get style properties on any component instance. The `UIObject.setStyle()` and `UIObject.getStyle()` methods can be called directly from any component. The following syntax specifies a property and value for a component instance:

```
instanceName.setStyle("propertyName", value);
```

For example, the following code sets the accent colors on a Button instance called `myButton` that uses the Halo theme:

```
myButton.setStyle("themeColor", "haloBlue");
```

Note: If the value is a string, it must be enclosed in quotation marks.

Even though you can access the styles directly as properties (for example, `myButton.color = 0xFF00FF`), it's best to use the `setStyle()` and `getStyle()` methods so that the styles work correctly and components are redrawn with the new style settings. For more information, see `UIObject.setStyle()` on page 825.

Style properties set on a component instance through `setStyle()` have the highest priority and override all other style settings discussed in the following sections.

Note: If you want to change multiple properties, or change properties for multiple component instances, you should create a custom style. For more information, see "Setting custom styles for groups of components" on page 69.

To set or change a property for a single component instance that uses the Halo theme:

1. Select the component instance on the Stage.
2. In the Property inspector, give it the instance name **myComponent**.
3. Open the Actions panel and select Scene 1, then select Layer 1: Frame 1.
4. Enter the following code to change the instance to orange:

   ```
   myComponent.setStyle("themeColor", "haloOrange");
   ```

5. Select Control > Test Movie to view the changes.

   For a list of styles supported by a particular component, see the component's entry in Chapter 6, "Components Dictionary," on page 91.

Setting custom styles for groups of components

You can create custom style declarations to specify a unique set of properties for groups of components in your Flash document. You create a new instance of the CSSStyleDeclaration object, create a custom style name and place it on the `_global.styles` list (`_global.styles.newStyle`), specify the properties and values for the style, and assign the style name to component instances that should share the same look.

To make changes to a custom style format, use the following syntax:

```
_global.styles.CustomStyleName.setStyle(propertyName, propertyValue);
```

The CSSStyleDeclaration object is accessible if you have placed at least one component instance on the Stage.

Custom style settings have priority over class, inherited, and global style settings.

For a list of the styles that each component supports, see the component entries in Chapter 6, "Components Dictionary," on page 91.

To create a custom style declaration for a group of components:

1. Make sure the document contains at least one component instance.

   For more information, see "Adding components to Flash documents" on page 44.

   This example uses three button components with the instance names `a`, `b`, and `c`. If you use different components, give them instance names in the Property inspector and use those instance names in step 9.
2. Create a new layer in the Timeline and give it a name.
3. Select a frame in the new layer on which (or before which) the component appears.
4. Open the Actions panel.
5. Use the following syntax to create an instance of the CSSStyleDeclaration object to define the new custom style format:

   ```
   var styleObj = new mx.styles.CSSStyleDeclaration();
   ```
6. Set the `styleName` property of the style declaration to name the style:

   ```
   styleObj.styleName = "newStyle";
   ```
7. Place the style on the `_global.styles` list:

   ```
   _global.styles.newStyle = styleObj;
   ```

   You can also create a CSSStyleDeclaration object and assign it to a new style declaration by using the following syntax:

   ```
   var styleObj = _global.styles.newStyle = new
     mx.styles.CSSStyleDeclaration();
   ```
8. Use the following syntax to specify the properties you want to define for the `newStyle` style declaration:

   ```
   styleObj.setStyle("fontFamily", "_sans");
   styleObj.setStyle("fontSize", 14);
   styleObj.setStyle("fontWeight", "bold");
   styleObj.setStyle("textDecoration", "underline");
   styleObj.setStyle("color", 0x336699);
   styleObj.setStyle("themeColor", "haloBlue");
   ```
9. In the same Script pane, use the following syntax to set the `styleName` property of three specific components to the custom style declaration:

   ```
   a.setStyle("styleName", "newStyle");
   b.setStyle("styleName", "newStyle");
   c.setStyle("styleName", "newStyle");
   ```

Now you can access styles on the custom style declaration using the `setStyle()` and `getStyle()` methods through its global `styleName` property. The following code sets the `backgroundColor` style on the `newStyle` style declaration:

```
_global.styles.newStyle.setStyle("themeColor", "haloOrange");
```

Another option in setting custom style declarations is you can assign the `CSSStyleDeclaration` instance directly to the component instance's `styleName` property and bypass storing the declaration in `_global.styles`. To use this approach, modify the above procedure as follows:

- Remove the ActionScript from steps 6 and 7 above.
- Modify the ActionScript in step 9 to assign the `CSSStyleDeclaration` instance directly to the component instances:

```
a.setStyle("styleName", styleObj);
b.setStyle("styleName", styleObj);
c.setStyle("styleName", styleObj);
```

Setting styles for a component class

You can define a class style declaration for any class of component (Button, CheckBox, and so on) that sets default styles for each instance of that class. You must create the style declaration before you create the instances. Some components, such as TextArea and TextInput, have class style declarations predefined by default because their `borderStyle` and `backgroundColor` properties must be customized.

Caution: If you replace a class style sheet, make sure to add any styles that you want from the old style sheet; otherwise, they will be overwritten.

The following code creates a class style declaration for CheckBox and sets the color for all check box instances to blue:

```
if (_global.styles.CheckBox == undefined) {
  _global.styles.CheckBox = new mx.styles.CSSStyleDeclaration();
}
_global.styles.CheckBox.setStyle("color", 0x0000FF);
```

Custom style settings have priority over inherited and global style settings.

Setting inheriting styles on a container

An *inheriting style* is a style that inherits its value from parent components in the document's MovieClip hierarchy. If a text or color style is not set at an instance, custom, or class level, Flash searches the MovieClip hierarchy for the style value. Thus, if you set styles on a container component, the contained components inherit these style settings.

The following styles are inheriting styles:

- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `textAlign`
- `textIndent`
- All single-value color styles (for example, `themeColor` is an inheriting style, but `alternatingRowColors` is not)

The Style Manager tells Flash whether a style inherits its value. Additional styles can also be added at runtime as inheriting styles. For more information, see "StyleManager class" on page 721.

Note: The CSS `inherit` value is not supported.

Inherited styles take priority over global styles.

The following example demonstrates how inheriting styles can be used with an Accordion component, which is available with Flash MX Professional. (The inheriting styles feature is supported by both Flash MX and Flash MX Professional.)

To create an Accordion component with styles that are inherited by the components in the individual Accordion panes:

1. Open a new FLA file.
2. Drag an Accordion component from the Components panel to the Stage.
3. Use the Property inspector to name and size the Accordion component. For this example, give the component the instance name **accordion**.
4. Drag a TextInput component and a Button component from the Components panel to the Stage. Select the instances on the Stage and delete them.

   By dragging the components to the Stage and immediately deleting them, you add the components to the library and make them available to your document at runtime.
5. Add the following ActionScript to the frame:

```
var section1 = accordion.createChild(mx.core.View, "section1", {label:
  "First Section"});
var section2 = accordion.createChild(mx.core.View, "section2", {label:
  "Second Section"});

var input1 = section1.createChild(mx.controls.TextInput, "input1");
var button1 = section1.createChild(mx.controls.Button, "button1");

input1.text = "Text Input";
button1.label = "Button";
button1.move(0, input1.height + 10);

var input2 = section2.createChild(mx.controls.TextInput, "input2");
var button2 = section2.createChild(mx.controls.Button, "button2");

input2.text = "Text Input";
button2.label = "Button";
button2.move(0, input2.height + 10);
```

   The above code adds two children to the Accordion component and loads each with a TextInput and Button control, which this example uses to demonstrate style inheritance.
6. Select Control > Test Movie to see the document before adding style inheritance.
7. Add the following ActionScript to the end of the frame:

```
accordion.setStyle("fontStyle", "italic");
```

8. Select Control > Test Movie to see the changes.

Notice that the `fontStyle` setting on the Accordion component affects not only the Accordion text itself but also the text associated with the TextInput and Button components inside the Accordion component.

Setting global styles

The global style declaration is assigned to all Flash components built with version 2 of the Macromedia Component Architecture. The _global object has a `style` property (`_global.style`) that is an instance of CSSStyleDeclaration. This property acts as the global style declaration. If you change a property's value on the global style declaration, the change is applied to all components in your Flash document.

Caution: Some styles are set on a component class's CSSStyleDeclaration instance (for example, the `backgroundColor` style of the TextArea and TextInput components). Because the class style declaration takes precedence over the global style declaration when style values are determined, setting `backgroundColor` on the global style declaration would have no effect on TextArea and TextInput components. For more information, see "Using global, custom, and class styles in the same document" on page 73.

To change one or more properties in the global style declaration:

1. Make sure the document contains at least one component instance.

   For more information, see "Adding components to Flash documents" on page 44.

2. Select a frame in the Timeline on which (or before which) the components appear.

3. In the Actions panel, use code like the following to change properties on the global style declaration. You need to list only the properties whose values you want to change, as shown here:

```
_global.style.setStyle("color", 0xCC6699);
_global.style.setStyle("themeColor", "haloBlue")
_global.style.setStyle("fontSize",16);
_global.style.setStyle("fontFamily" , "_serif");
```

4. Select Control > Test Movie to see the changes.

Using global, custom, and class styles in the same document

If you define a style in only one place in a document, Flash uses that definition when it needs to know a property's value. However, one Flash document can have a variety of style settings—style properties set directly on component instances, custom style declarations, default class style declarations, inheriting styles, and a global style declaration. In such a situation, Flash determines the value of a property by looking for its definition in all these places in a specific order.

Flash looks for styles in the following order until a value is found:

1. Flash looks for a style property on the component instance.
2. Flash looks at the `styleName` property of the instance to see if a custom style declaration is assigned to it.
3. Flash looks for the property on a default class style declaration.

4. If the style is one of the inheriting styles, Flash looks through the parent hierarchy for an inherited value.
5. Flash looks for the style in the global style declaration.
6. If the property is still not defined, the property has the value `undefined`.

About color style properties

Color style properties behave differently than noncolor properties. All color properties have a name that ends in "Color"—for example, `backgroundColor`, `disabledColor`, and `color`. When color style properties are changed, the color is immediately changed on the instance and in all of the appropriate child instances. All other style property changes simply mark the object as needing to be redrawn, and changes don't occur until the next frame.

The value of a color style property can be a number, a string, or an object. If it is a number, it represents the RGB value of the color as a hexadecimal number (0xRRGGBB). If the value is a string, it must be a color name.

Color names are strings that map to commonly used colors. You can add new color names by using the Style Manager (see "StyleManager class" on page 721). The following table lists the default color names:

Color name	Value
black	0x000000
white	0xFFFFFF
red	0xFF0000
green	0x00FF00
blue	0x0000FF
magenta	0xFF00FF
yellow	0xFFFF00
cyan	0x00FFFF
haloGreen	0x80FF4D
haloBlue	0x2BF5F5
haloOrange	0xFFC200

Note: If the color name is not defined, the component may not draw correctly.

You can use any valid ActionScript identifier to create your own color names (for example, `"WindowText"` or `"ButtonText"`). Use the Style Manager to define new colors, as shown here:

```
mx.styles.StyleManager.registerColorName("special_blue", 0x0066ff);
```

Most components cannot handle an object as a color style property value. However, certain components can handle color objects that represent gradients or other color combinations. For more information, see the "Using styles" section of each component's entry in Chapter 6, "Components Dictionary."

You can use class style declarations and color names to easily control the colors of text and symbols on the screen. For example, if you want to provide a display configuration screen that looks like Microsoft Windows, you would define color names like ButtonText and WindowText and class style declarations like Button, CheckBox, and Window.

Note: Some components provide style properties that are an array of colors, such as `alternatingRowColors`. You must set these styles only as an array of numeric RGB values, not color names.

Customizing component animations

Several components, such as the Accordion, ComboBox, and Tree components, provide animation to demonstrate the transition between component states—for example, when switching between Accordion children, expanding the ComboBox drop-down list, and expanding or collapsing Tree folders. Additionally, components provide animation related to the selection and deselection of an item, such as rows in a list.

You can control aspects of these animations through the following styles:

Animation style	Description
`openDuration`	The duration of the transition for open easing in Accordion, ComboBox, and Tree components, in milliseconds. The default value is 250.
`openEasing`	A reference to a tweening function that controls the state animation in the Accordion, ComboBox, and Tree components. The default equation uses a sine in/out formula.
`popupDuration`	The duration of the transition as a menu opens in the Menu component, in milliseconds. The default value is 150. Note, however, that the animation always uses the default sine in/out equation.
`selectionDuration`	The duration of the transition in ComboBox, DataGrid, List, and Tree components from a normal to selected state or back from selected to normal, in milliseconds. The default value is 200.
`selectionEasing`	A reference to a tweening function that controls the selection animation in ComboBox, DataGrid, List, and Tree components. This style applies only for the transition from a normal to a selected state. The default equation uses a sine in/out formula.

The mx.transtions.easing package provides six classes to control easing:

Easing class	Description
`Back`	Extends beyond the transition range at one or both ends one time to provide a slight overflow effect.
`Bounce`	Provides a bouncing effect entirely within the transition range at one or both ends. The number of bounces is related to the duration: longer durations produce more bounces.
`Elastic`	Provides an elastic effect that falls outside the transition range at one or both ends. The amount of elasticity is unaffected by the duration.

Easing class	Description
`None`	Provides an equal movement from start to end with no effects, slowing, or speeding. This transition is also commonly referred to as a *linear transition*.
`Regular`	Provides for slower movement at one or both ends for a speeding-up effect, a slowing-down effect, or both.
`Strong`	Provides for much slower movement at one or both ends. This effect is similar to Regular but is more pronounced.

Each of the classes in the mx.transitions.easing package provides the following three easing methods:

Easing method	Description
`easeIn`	Provides the easing effect at the beginning of the transition.
`easeOut`	Provides the easing effect at the end of the transition.
`easeInOut`	Provides the easing effect at the beginning and end of the transition.

Because the easing methods are static methods of the easing classes, you never need to instantiate the easing classes. The methods are used in calls to `setStyle()`, as in the following example.

```
import mx.transitions.easing.*;
trace("_global.styles.Accordion = " + _global.styles.Accordion);
_global.styles.Accordion.setStyle("openDuration", 1500);
_global.styles.Accordion.setStyle("openEasing", Bounce.easeOut);
```

Note: The default equation used by all transitions is not available in the easing classes listed above. To specify that a component should use the default easing method after another easing method has been specified, call `setStyle("openEasing", null)`.

Getting style property values

To retrieve a style property value, use `UIObject.getStyle()`. Every component that is a subclass of UIObject (which includes all version 2 components except the Media components) inherits the `getStyle()` method. This means you can call `getStyle()` from any component instance, just as you can call `setStyle()` from any component instance.

The following code gets the value of the `themeColor` style and assigns it to the variable `oldStyle`:

```
var myCheckBox:mx.controls.CheckBox;
var oldFontSize:Number

oldFontSize = myCheckBox.getStyle("fontSize");
trace(oldFontSize);
```

About themes

Themes are collections of styles and skins. The default theme for Flash MX 2004 and Flash MX Professional 2004 is called Halo (HaloTheme.fla). The Halo theme lets you provide a responsive, expressive experience for your users. Flash MX 2004 and Flash MX Professional 2004 include one additional theme, Sample (SampleTheme.fla). The Sample theme provides an example of how you can use more styles for customization. (The Halo theme does not use all styles included in the Sample theme.) The theme files are located in the following folders in a default installation:

- Windows: \Program Files\Macromedia\Flash MX 2004*language*\Configuration\ComponentFLA\
- Macintosh: HD/Applications/Macromedia Flash MX 2004/Configuration/ComponentFLA/

You can create new themes and apply them to an application to change the look and feel of all the components. For example, you could create themes that mimic the native operating system appearance.

Components use skins (graphic or movie clip symbols) to display their appearances. The AS file that defines each component contains code that loads specific skins for the component. You can easily create a new theme by making a copy of the Halo or Sample theme and altering the graphics in the skins.

A theme can also contain a new set of style default values. You must write ActionScript code to create a global style declaration and any additional style declarations. For more information, see "Modifying default style property values in a theme" on page 78.

Creating a new theme

If you don't want to use the Halo theme or the Sample theme, you can modify one of them to create a new theme.

Some skins in the themes have a fixed size. You can make them larger or smaller and the components will automatically resize to match them. Other skins are composed of multiple pieces, some static and some that stretch.

Some skins (for example, RectBorder and ButtonSkin) use the ActionScript drawing API to draw their graphics, because it is more efficient in terms of size and performance. You can use the ActionScript code in those skins as a template to adjust the skins to your needs.

For a list of the skins supported by each component and their properties, see Chapter 6, "Components Dictionary," on page 91.

To create a new theme:

1. Select the theme FLA file that you want to use as a template, and make a copy.

   Give the copy a unique name such as **MyTheme.fla**.
2. Select File > Open MyTheme.fla in Flash.
3. Select Window > Library to open the library if it isn't open already.

4. Double-click any skin symbol you want to modify to open it in symbol-editing mode.

   The skins are located in the Flash UI Components 2/Themes/MMDefault/*Component* Assets folder (this example uses RadioButton Assets).

5. Modify the symbol or delete the graphics and create new graphics.

   You may need to select View > Zoom In to increase the magnification. When you edit a skin, you must maintain the registration point in order for the skin to be displayed correctly. The upper left corner of all edited symbols must be at (0,0).

   For example, open the States/RadioFalseDisabled asset and change the inner circle to a light gray.

6. When you finish editing the skin symbol, click the Back button at the left side of the information bar at the top of the Stage to return to document-editing mode.

7. Repeat steps 4-6 until you've edited all the skins you want to change.

8. Apply MyTheme.fla to a document by following the steps shown later in this chapter. (See "Applying a theme to a document" on page 79.)

Modifying default style property values in a theme

The default style property values are provided by each theme in a class named Default. To change the defaults for a custom theme, create a new ActionScript class called Default in a package appropriate for your theme, and change the default settings as desired.

To modify default style values in a theme:

1. Create a new folder for your theme in First Run/Classes/mx/skins.

   For example, create a folder called **myTheme**.

2. Copy an existing Defaults class to your new theme folder.

   For example, copy mx/skins/halo/Defaults.as to mx/skins/myTheme/Defaults.as.

3. Open the new Defaults class in an ActionScript editor.

   Flash MX 2004 Professional users can open the file within Flash MX 2004 Professional. Flash MX 2004 users can open the file in Notepad in Windows or SimpleText on the Macintosh.

4. Modify the class declaration to reflect the new package.

   For example, our new class declaration is `class mx.skins.myTheme.Defaults`.

5. Modify the style settings as desired.

   For example, change the default disabled color to a dark red.

   ```
   o.disabledColor = 0x663333;
   ```

6. Save the changed Defaults class file.

7. Copy an existing FocusRect class from the source theme to your custom theme.

   For example, copy mx/skins/halo/FocusRect.as to mx/skins/myTheme/FocusRect.as.

8. Open the new FocusRect class in an ActionScript editor.

9. Modify all references to the source theme's package to the new theme's package.

   For example, change all occurrences of "halo" to "myTheme".

10. Save the changed FocusRect class file.
11. Open the FLA file for your custom theme.

    This example uses MyTheme.fla.

12. Open the library (Window > Library) and locate the Defaults symbol.

    In this example, it's in Flash UI Components 2/Themes/MMDefault/Defaults.

13. Edit the symbol properties for the Default symbol.
14. Change the AS 2.0 Class setting to reflect your new package.

    The example class is mx.skins.myTheme.Defaults.

15. Click OK.
16. Locate the FocusRect symbol.

    In this example, it's in Flash UI Components 2/Themes/MMDefault/FocusRect.

17. Edit the symbol properties for the FocusRect symbol.
18. Change the AS 2.0 Class setting to reflect your new package.

    The example class is mx.skins.myTheme.FocusRect.

19. Click OK.
20. Apply the custom theme to a document by following the steps in the next section.

    Remember to include the Defaults and FocusRect symbols when dragging assets from your custom theme to the target document.

In this example you used a new theme to customize the text color of disabled components. This particular customization, changing a single default style property value, would have been accomplished more easily through styling as explained in "Using styles to customize component color and text" on page 67. Using a new theme to customize defaults is appropriate when customizing many style properties or when already creating a new theme to customize component graphics.

Applying a theme to a document

To apply a new theme to a document, open a theme FLA file as an external library, and drag the theme folder from the external library to the document library. The following steps explain the process in detail.

To apply a theme to a document:

1. Select File > Open and open the document that uses version 2 components in Flash, or select File > New and create a new document that uses version 2 components.
2. Select File > Save and choose a unique name such as **ThemeApply.fla**.

3. Select File > Import > Open External Library, and select the FLA file of the theme you want to apply to your document.

   If you haven't created a new theme, you can use the Sample theme.

4. In the theme's Library panel, select Flash UI Components 2/Themes/MMDefault and drag the Assets folder of any components in your document to the ThemeApply.fla library.

   For example, drag the RadioButton Assets folder to the ThemeApply.fla library.

   If you're unsure about which components are in the document, drag the Sample Theme movie clip to the Stage. The skins are automatically assigned to components in the document.

   Note: The Live Preview of the components on the Stage will not reflect the new theme.

5. If you dragged individual component Assets folders to the ThemeApply.fla library, make sure the Assets symbol for each component is set to Export in First Frame.

   For example, the Assets folder for the RadioButton component is called RadioButton Assets; it has a symbol called RadioButtonAssets, which contains all of the individual asset symbols. If you set Export in First Frame on the RadioButtonAssets symbol, all individual asset symbols will also export in the first frame.

6. Select Control > Test Movie to see the document with the new theme applied.

   In this example, make sure you have a RadioButton instance on the Stage and set its `enabled` property to `false` in the Actions panel in order to see the new disabled RadioButton appearance.

About skinning components

Skins are movie clip symbols a component uses to display its appearance. Most skins contain shapes that represent the component's appearance. Some skins contain only ActionScript code that draws the component in the document.

Version 2 components are compiled clips—you cannot see their assets in the library. However, the Flash installation includes FLA files that contain all the component skins. These FLA files are called *themes*. Each theme has a different appearance and behavior, but contains skins with the same symbol names and linkage identifiers. This lets you drag a theme onto the Stage in a document to change its appearance. You also use the theme FLA files to edit component skins. The skins are located in the Themes folder in the Library panel of each theme FLA file. (For more information about themes, see "About themes" on page 77.)

Each component is composed of many skins. For example, the down arrow of the ScrollBar subcomponent consists of four skins: ScrollDownArrowDisabled, ScrollDownArrowDown, ScrollDownArrowOver, and ScrollDownArrowUp. The entire ScrollBar uses 13 different skin symbols.

Some components share skins; for example, components that use scroll bars—such as ComboBox, List, and ScrollPane—share the skins in the ScrollBar Skins folder. You can edit existing skins and create new skins to change the appearance of components.

The AS file that defines each component class contains code that loads specific skins for the component. Each component skin corresponds to a skin property that is assigned to a skin symbol's linkage identifier. For example, the pressed (down) state of the down arrow of the ScrollBar component has the skin property name `downArrowDownName`. The default value of the `downArrowDownName` property is `"ScrollDownArrowDown"`, which is the linkage identifier of the skin symbol in the theme FLA file. You can edit existing skins and apply them to all components that use the skin by editing the skin symbol and leaving the existing linkage identifier. You can create new skins and apply them to specific component instances by setting the skin properties for a component instance. You do not need to edit the component's AS file to change its skin properties; you can pass skin property values to the component's constructor function when the component is created in your document.

The skin properties for each component are listed in each component's entry in the Components Dictionary. For example, the skin properties for the Button component are located here: Components Dictionary > Button component > Customizing the Button component > Using skins with the Button component.

Choose one of the following ways to skin a component according to what you want to do. These approaches are listed from easiest to most difficult.

- To change the skins associated with all instances of a particular component in a single document, copy and modify individual skin elements. (See "Editing component skins in a document" on page 81).

  This method of skinning is recommended for beginners, because it doesn't require any scripting.
- To replace all the skins in a document with a new set (with each kind of component sharing the same appearance), apply a theme. (See "About themes" on page 77.)

  This method of skinning is recommended for applying a consistent look and feel across all components and across several documents.
- To link the color of a skin element to a style property, add ActionScript code to the skin to register it as a colored skin element. (See "Linking skin color to styles" on page 83).
- To use different skins for multiple instances of the same component, create new skins and set skin properties. (See "Creating new component skins" on page 83 and "Applying new skins to a component" on page 85.)
- To change skins in a subcomponent (such as a scroll bar in a List component), subclass the component. (See "Applying new skins to a subcomponent" on page 86.)
- To change skins of a subcomponent that aren't directly accessible from the main component (such as a List component in a ComboBox component), replace skin properties in the prototype. (See "Changing skin properties in the prototype" on page 89.)

Editing component skins in a document

To edit the skins associated with all instances of a particular component in a single document, copy the skin symbols from the theme to the document and edit the graphics as desired.

The procedure described below is very similar to creating and applying a new theme (see "About themes" on page 77). The primary difference is that this procedure describes copying symbols directly from the theme already in use to a single document and editing only a small number of all skins available. This is appropriate when your modifications are all in a single document and when you are modifying skins for only a few components. If the edited skins will be shared in multiple documents or encompass changes in several components, you may find it maintenance to be easier if you create a new theme.

To edit component skins in a document:

1. If you already applied the Sample theme to a document, skip to step 5.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, select Flash UI Components 2/Themes/MMDefault and drag the Assets folder of any components in your document to the library for your document.

   For example, drag the RadioButton Assets folder to the ThemeApply.fla library.
4. If you dragged individual component Assets folders to the library, make sure the Assets symbol for each component is set to Export in First Frame.

   For example, the Assets folder for the RadioButton component is called RadioButton Assets; it has a symbol called RadioButtonAssets, which contains all of the individual asset symbols. If you set Export in First Frame on the RadioButtonAssets symbol, all individual asset symbols will also export in the first frame.
5. Double-click any skin symbol you want to modify to open it in symbol-editing mode.

   For example, open the States/RadioFalseDisabled symbol.
6. Modify the symbol or delete the graphics and create new graphics.

   You may need to select View > Zoom In to increase the magnification. When you edit a skin, you must maintain the registration point in order for the skin to be displayed correctly. The upper left corner of all edited symbols must be at (0,0).

   For example, change the inner circle to a light gray.
7. When you finish editing the skin symbol, click the Back button at the left side of the information bar at the top of the Stage to return to document-editing mode.
8. Repeat steps 5-7 until you've edited all the skins you want to change.

   Note: The live preview of the components on the Stage will not reflect the edited skins.
9. Select Control > Test Movie.

   In this example, make sure you have a RadioButton instance on the Stage and set its `enabled` property to `false` in the Actions panel in order to see the new disabled RadioButton appearance.

Creating new component skins

If you want to use a particular skin for one instance of a component, but another skin for another instance of the component, you must open a theme FLA file and create a new skin symbol. Components are designed to make it easy to use different skins for different instances.

To create a new skin:

1. Select File > Open and open the theme FLA file that you want to use as a template.
2. Select File > Save As and select a unique name, such as **MyTheme.fla**.
3. Select the skins that you want to edit (in this example, RadioTrueUp).

   The skins are located in the Themes/MMDefault/*Component* Assets folder (in this example, Themes/MMDefault/RadioButton Assets/States).
4. Select Duplicate from the Library options menu (or by right-clicking the symbol), and give the symbol a unique name, such as **MyRadioTrueUp**.
5. Click Advanced in the Symbol Properties dialog box, and select Export for ActionScript.

   A linkage identifier that matches the symbol name is entered automatically.
6. Double-click the new skin in the library to open it in symbol-editing mode.
7. Modify the movie clip, or delete it and create a new one.

   You may need to select View > Zoom In to increase the magnification. When you edit a skin, you must maintain the registration point in order for the skin to be displayed correctly. The upper left corner of all edited symbols must be at (0,0).
8. When you finish editing the skin symbol, click the Back button at the left side of the information bar at the top of the Stage to return to document-editing mode.
9. Select File > Save but don't close MyTheme.fla. Now you must create a new document in which to apply the edited skin to a component.

   For more information, see "Applying new skins to a component" on page 85, "Applying new skins to a subcomponent" on page 86, or "Changing skin properties in the prototype" on page 89.

Note: Flash does not display changes made to component skins when you view components on the Stage using Live Preview.

Linking skin color to styles

The version 2 component framework makes it easy to link a visual asset in a skin element to a style set on the component using the skin. To register a movie clip instance to a style, or an entire skin element to a style, add ActionScript code in the Timeline of the skin to call `mx.skins.ColoredSkinElement.setColorStyle(`*`targetMovieClip`*`, `*`styleName`*`)`.

To link a skin to a style property:

1. If you already applied the Sample theme to a document, skip to step 5.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, select Flash UI Components 2/Themes/MMDefault, and drag the Assets folder of any components in your document to the library for your document.

   For example, drag the RadioButton Assets folder to the target library.
4. If you dragged individual component assets folders to the library, make sure the Assets symbol for each component is set to Export in First Frame.

   For example, the Assets folder for the RadioButton component is called RadioButton Assets; it has a symbol called RadioButtonAssets, which contains all of the individual asset symbols. If you set Export in First Frame on the RadioButtonAssets symbol, all individual asset symbols will also export in the first frame.
5. Double-click any skin symbol you want to modify to open it in symbol-editing mode.

   For example, open the States/RadioFalseDisabled symbol.
6. If the element to be colored is a graphic symbol and not a movie clip instance, use Modify > Convert to Symbol to covert it to a movie clip instance.

   For this example, change the center graphic, which is an instance of the graphic symbol RadioShape1, to a movie clip symbol; then name it **Inner Circle**. You do not need to select Export for ActionScript.

   It would be good practice, but it is not required, to move the newly created movie clip symbol to the Elements folder of the component assets being edited.
7. If you converted a graphic symbol to a movie clip instance in the previous step, give that instance a name so it can be targeted in ActionScript.

   For this example, name the instance **innerCircle**.
8. Add ActionScript code to register the skin element or a movie clip instance it contains as a colored skin element.

   For example, add the following code to the skin element's Timeline.

   ```
   mx.skins.ColoredSkinElement.setColorStyle(innerCircle,
     "symbolBackgroundDisabledColor");
   ```

   In this example you're using a color that already corresponds to an existing style name in the Sample style. Wherever possible, it's best to use style names corresponding to official Cascading Style Sheet standards or styles provided by the Halo and Sample themes.
9. Repeat steps 5-8 until you've edited all the skins you want to change.

   For this example, repeat these steps for the RadioTrueDisabled skin, but instead of converting the existing graphic to a movie clip, delete the graphic and drag the existing Inner Circle symbol to the RadioTrueDisabled skin element.
10. When you finish editing the skin symbol, click the Back button at the left side of the information bar at the top of the Stage to return to document-editing mode.

11. Drag an instance of the component to the Stage.

    For this example, drag two RadioButton components to the Stage, set one to selected, and use ActionScript to set both to disabled in order to see the changes.

12. Add ActionScript code to the document to set the new style property on the component instances or at the global level.

    For this example, set the property at the global level as follows:

    ```
    _global.style.setStyle("symbolBackgroundDisabledColor", 0xD9D9D9);
    ```

13. Select Control > Test Movie.

Applying new skins to a component

Once you have created a new skin, you must apply it to a component in a document. You can use the `createClassObject()` method to dynamically create the component instances, or you can manually place the component instances on the Stage. There are two different ways to apply skins to component instances, depending on how you add the components to a document.

To dynamically create a component and apply a new skin:

1. Select File > New to create a new Flash document.
2. Select File > Save and give the file a unique name, such as **DynamicSkinning.fla**.
3. Drag any components from the Components panel to the Stage, including the component whose skin you edited (in this example, RadioButton), and delete them.

   This adds the symbols to the document's library, but doesn't make them visible in the document.

4. Drag MyRadioTrueUp and any other symbols you customized from MyTheme.fla to the Stage of DynamicSkinning.fla, and delete them.

   This adds the symbols to the document's library, but doesn't make them visible in the document.

5. Open the Actions panel and enter the following on Frame 1:

   ```
   import mx.controls.RadioButton;
   createClassObject(RadioButton, "myRadio", 0, {trueUpIcon:"MyRadioTrueUp",
     label: "My Radio Button"});
   ```

6. Select Control > Test Movie.

To manually add a component to the Stage and apply a new skin:

1. Select File > New to create a new Flash document.
2. Select File > Save and give the file a unique name, such as **ManualSkinning.fla**.
3. Drag components from the Components panel to the Stage, including the component whose skin you edited (in this example, RadioButton).
4. Drag MyRadioTrueUp and any other symbols you customized from MyTheme.fla to the Stage of ManualSkinning.fla, and delete them.

   This adds the symbols to the document's library, but doesn't make them visible in the document.

5. Select the RadioButton component on the Stage and open the Actions panel.
6. Attach the following code to the RadioButton instance:

```
onClipEvent(initialize){
  trueUpIcon = "MyRadioTrueUp";
}
```

7. Select Control > Test Movie.

Applying new skins to a subcomponent

In certain situations you may want to modify the skins of a subcomponent in a component, but the skin properties are not directly available (for example, there is no direct way to alter the skins of the scroll bar in a List component). The following code lets you access the scroll bar skins. All the scroll bars that are created after this code runs will also have the new skins.

If a component is composed of subcomponents, the subcomponents are identified in the component's entry in Chapter 6, "Components Dictionary."

To apply a new skin to a subcomponent:

1. Follow the steps in "Creating new component skins" on page 83, but edit a scroll bar skin. For this example, edit the ScrollDownArrowDown skin and give it the new name **MyScrollDownArrowDown**.
2. Select File > New to create a new Flash document.
3. Select File > Save and give the file a unique name, such as **SubcomponentProject.fla**.
4. Double-click the List component in the Components panel to add it to the Stage, and press Backspace to delete it from the Stage.

   This adds the component to the Library panel, but doesn't make the component visible in the document.
5. Drag MyScrollDownArrowDown and any other symbols you edited from MyTheme.fla to the Stage of SubcomponentProject.fla, and delete them.

   This adds the symbol to the Library panel, but doesn't make it visible in the document.
6. Do one of the following:

   - If you want to change all scroll bars in a document, enter the following code in the Actions panel on Frame 1 of the Timeline:

     ```
     import mx.controls.List;
     import mx.controls.scrollClasses.ScrollBar;
     ScrollBar.prototype.downArrowDownName = "MyScrollDownArrowDown";
     ```

     You can then enter the following code on Frame 1 to create a list dynamically:

     ```
     createClassObject(List, "myListBox", 0, {dataProvider: ["AL","AR","AZ",
       "CA","HI","ID", "KA","LA","MA"]});
     ```

     Or, you can drag a List component from the library to the Stage.

- If you want to change a specific scroll bar in a document, enter the following code in the Actions panel on Frame 1 of the Timeline:

```
import mx.controls.List
import mx.controls.scrollClasses.ScrollBar
var oldName = ScrollBar.prototype.downArrowDownName;
ScrollBar.prototype.downArrowDownName = "MyScrollDownArrowDown";
createClassObject(List, "myList1", 0, {dataProvider: ["AL","AR","AZ",
  "CA","HI","ID", "KA","LA","MA"]});
myList1.redraw(true);
ScrollBar.prototype.downArrowDownName = oldName;
```

  Note: Set enough data so that the scroll bars appear, or set the `vScrollPolicy` property to `true`.

7. Select Control > Test Movie.

You can also set subcomponent skins for all components in a document by setting the skin property on the subcomponent's `prototype` object in the `#initclip` section of a skin symbol.

To use #initclip to apply an edited skin to all components in a document:

1. Follow the steps in "Creating new component skins" on page 83, but edit a scroll bar skin. For this example, edit the ScrollDownArrowDown skin and give it the new name **MyScrollDownArrowDown**.
2. Select File > New and create a new Flash document. Save it with a unique name, such as **SkinsInitExample.fla**.
3. Select the MyScrollDownArrowDown symbol from the library of the edited theme library example, drag it to the Stage of SkinsInitExample.fla, and delete it.

   This adds the symbol to the library without making it visible on the Stage.
4. Select MyScrollDownArrowDown in the SkinsInitExample.fla library, and select Linkage from the Library options menu.
5. Select the Export for ActionScript check box. Click OK.

   Export in First Frame is automatically selected.
6. Double-click MyScrollDownArrowDown in the library to open it in symbol-editing mode.
7. Enter the following code on Frame 1 of the MyScrollDownArrowDown symbol:

```
#initclip 10
  import mx.controls.scrollClasses.ScrollBar;
  ScrollBar.prototype.downArrowDownName = "MyScrollDownArrowDown";
#endinitclip
```

8. Do one of the following to add a List component to the document:
   - Drag a List component from the Components panel to the Stage. Enter enough label parameters so that the vertical scroll bar will appear.
   - Drag a List component from the Components panel to the Stage and delete it. Enter the following code on Frame 1 of the main Timeline of SkinsInitExample.fla:

```
createClassObject(mx.controls.List, "myListBox1", 0, {dataProvider:
  ["AL","AR","AZ", "CA","HI","ID", "KA","LA","MA"]});
```

     Note: Add enough data so that the vertical scroll bar appears, or set `vScrollPolicy` to `true`.

The following example explains how to skin something that's already on the Stage. This example skins only List scroll bars; any TextArea or ScrollPane scroll bars would not be skinned.

To use #initclip to apply an edited skin to specific components in a document:

1. Follow the steps in "Editing component skins in a document" on page 81, but edit a scroll bar skin. For this example, edit the ScrollDownArrowDown skin and give it the new name **MyScrollDownArrowDown**.
2. Select File > New and create a Flash document.
3. Select File > Save and give the file a unique name, such as **MyVScrollTest.fla.**
4. Drag MyScrollDownArrowDown from the theme library to the MyVScrollTest.fla library.
5. Select Insert > New Symbol and give the symbol a unique name, such as **MyVScrollBar**.
6. Select the Export for ActionScript check box. Click OK.

   Export in First Frame is automatically selected.
7. Enter the following code on Frame 1 of the MyVScrollBar symbol:

```
#initclip 10
  import MyVScrollBar
  Object.registerClass("VScrollBar", MyVScrollBar);
#endinitclip
```

8. Drag a List component from the Components panel to the Stage.
9. In the Property inspector, enter as many Label parameters as necessary for the vertical scroll bar to appear.
10. Select File > Save.
11. Select File > New and create a new ActionScript file.
12. Enter the following code:

```
import mx.controls.VScrollBar
import mx.controls.List
class MyVScrollBar extends VScrollBar{
  function init():Void{
    if (_parent instanceof List){
      downArrowDownName = "MyScrollDownArrowDown";
    }
    super.init();
  }
}
```

13. Select File > Save and save this file as **MyVScrollBar.as**.
14. Click a blank area on the Stage and, in the Property inspector, click the Publish Settings button.
15. Click the ActionScript Version Settings button.
16. Click the Add New Path (+) button to add a new classpath, and select the Target button to browse to the location of the MyVScrollBar.as file on your hard disk.
17. Select Control > Test Movie.

Changing skin properties in the prototype

If a component does not directly support skin variables, you can subclass the component and replace its skins. For example, the ComboBox component doesn't directly support skinning its drop-down list, because the ComboBox component uses a List component as its drop-down list.

If a component is composed of subcomponents, the subcomponents are identified in the component's entry in Chapter 6, "Components Dictionary."

To skin a subcomponent:

1. Follow the steps in "Editing component skins in a document" on page 81, but edit a scroll bar skin. For this example, edit the ScrollDownArrowDown skin and give it the new name **MyScrollDownArrowDown**.
2. Select File > New and create a Flash document.
3. Select File > Save and give the file a unique name, such as **MyComboTest.fla.**
4. Drag MyScrollDownArrowDown from the theme library above to the Stage of MyComboTest.fla, and delete it.

   This adds the symbol to the library, but doesn't make it visible on the Stage.
5. Select Insert > New Symbol and give the symbol a unique name, such as **MyComboBox**.
6. Select the Export for ActionScript check box and click OK.

   Export in First Frame is automatically selected.
7. Enter the following code in the Actions panel on Frame 1 of the MyComboBox symbol:

```
#initclip 10
  import MyComboBox
  Object.registerClass("ComboBox", MyComboBox);
#endinitclip
```

8. When you finish editing the symbol, click the Back button at the left side of the information bar at the top of the Stage to return to document-editing mode.
9. Drag a ComboBox component to the Stage.
10. In the Property inspector, enter as many Label parameters as necessary for the vertical scroll bar to appear.
11. Select File > Save.
12. Select File > New and create a new ActionScript file.

13. Enter the following code:

```
import mx.controls.ComboBox
import mx.controls.scrollClasses.ScrollBar
class MyComboBox extends ComboBox{
  function getDropdown():Object{
    var oldName = ScrollBar.prototype.downArrowDownName;
    ScrollBar.prototype.downArrowDownName = "MyScrollDownArrowDown";
    var r = super.getDropdown();
    ScrollBar.prototype.downArrowDownName = oldName;
    return r;
  }
}
```

14. Select File > Save and save this file as **MyComboBox.as**.
15. Return to the file MyComboTest.fla.
16. Click a blank area on the Stage and, in the Property inspector, click the Publish Settings button.
17. Click the ActionScript Version Settings button.
18. Click the Add New Path (+) button to add a new classpath, and select the Target button to browse to the location of the MyComboBox.as file on your hard disk.
19. Select Control > Test Movie.

CHAPTER 6
Components Dictionary

This reference chapter describes each component and its application programming interface (API). Each component description contains information about the following:

- Keyboard interaction
- Live preview
- Accessibility
- Setting the component parameters
- Using the component in an application
- Customizing the component with styles and skins
- ActionScript methods, properties, and events

Components are presented alphabetically. You can also find components arranged by category in the tables that follow.

This chapter contains the following sections:

Types of components

The following tables list the different components, arranged by category, in version 2 of the Macromedia Component Architecture.

User interface (UI) components

Component	Description
Accordion component (Flash Professional only)	A set of vertical overlapping views with buttons along the top that allow users to switch views.
Alert component (Flash Professional only)	A window that presents a message and buttons to capture the user's response.
Button component	A resizable button that can be customized with a custom icon.

Component	Description
CheckBox component	Allows users to make a Boolean (true or false) choice.
ComboBox component	Allows users to select one option from a scrolling list of choices. This component can have an selectable text field at the top of the list that allows users to search the list.
DataGrid component (Flash Professional only)	Allows users to display and manipulate multiple columns of data.
DateChooser component (Flash Professional only)	Allows users to select one or more dates from a calendar.
DateField component (Flash Professional only)	An nonselectable text field with a calendar icon. When a user clicks inside the component's bounding box, Macromedia Flash displays a DateChooser component.
Label component	A non-editable, single-line text field.
List component	Allows users to select one or more options from a scrolling list.
Loader component	A container that holds a loaded SWF or JPEG file.
Menu component (Flash Professional only)	A standard desktop application menu; allows users to select one command from a list.
MenuBar component (Flash Professional only)	A horizontal bar of menus.
NumericStepper component	A text box with clickable arrows that raise and lower the value of a number.
ProgressBar component	Displays the progress of a process, such as a loading operation.
RadioButton component	Allows users to select between mutually exclusive options.
ScrollPane component	Displays movies, bitmaps, and SWF files in a limited area using automatic scroll bars.
TextArea component	An optionally editable, multiline text field.
TextInput component	An optionally editable, single-line text input field.
Tree component (Flash Professional only)	Allows a user to manipulate hierarchical information.
Window component	A draggable window with a title bar, caption, border, and Close button and content-display area.
UIScrollBar component	Allows you to add a scroll bar to a text field.

Data handling

Component	Description
Data binding classes (Flash Professional only)	Classes that implement the Flash runtime data binding functionality.
DataHolder component (Flash Professional only)	Holds data and can be used as a connector between components.

Component	Description
DataProvider API	The model for linear-access lists of data; it provides simple array-manipulation capabilities that broadcast data changes.
DataSet component (Flash Professional only)	A building block for creating data-driven applications.
RDBMSResolver component (Flash Professional only)	Lets you save data back to any supported data source. This component translates the XML that can be received and parsed by a web service, JavaBean, servlet, or ASP page.
Web service classes (Flash Professional only)	Classes that allow access to web services that use Simple Object Access Protocol (SOAP). These classes are in the mx.services package.
WebServiceConnector component (Flash Professional only)	Provides scriptless access to web service method calls.
XMLConnector component (Flash Professional only)	Reads and writes XML documents by using the HTTP `GET` and `POST` methods.
XUpdateResolver component (Flash Professional only)	Lets you save data back to any supported data source. This component translates the delta packet into XUpdate.

Media components

Component	Description
MediaController component	Controls streaming media playback in an application.
MediaDisplay component	Displays streaming media in an application.
MediaPlayback component	A combination of the MediaDisplay and MediaController components.

For more information on these components, see "Media components (Flash Professional only)" on page 497.

Managers

Class	Description
DepthManager class	Manages the stacking depths of objects.
FocusManager class	Handles Tab key navigation between components. Also handles focus changes as users click in the application.
PopUpManager class	Lets you create and delete pop-up windows.
StyleManager class	Lets you register styles and manages inherited styles.
SystemManager class	Lets you manage which top-level window is activated.

Screens

Class	Description
Form class (Flash Professional only)	Lets you manipulate form application screens at runtime.
Screen class	Base class for the Slide and Form classes. See Screen class (Flash Professional only).
Slide class (Flash Professional only)	Lets you manipulate slide presentation screens at runtime.

Other listings in this chapter

This chapter also describes several classes and APIs that don't fall into the above categories of components. They are listed in the following table.

Item	Description
CellRenderer API	A set of properties and methods that the list-based components (List, DataGrid, Tree, Menu, and ComboBox) use to manipulate and display custom cell content for each of their rows.
Collection interface (Flash Professional only)	Lets you manage a group of related items, called collection items. Each collection item in this set has properties that are described in the metadata of the collection item class definition.
DataGridColumn class (Flash Professional only)	Lets you create objects to use as columns of a data grid.
Delegate class	Allows a function passed from one object to another to be run in the context of the first object.
Delta interface (Flash Professional only)	Provides access to the transfer object, collection, and transfer object-level changes.
DeltaItem class (Flash Professional only)	Provides information about an individual operation performed on a transfer object.
DeltaPacket interface (Flash Professional only)	Along with the Delta interface and DeltaItem class, lets you manage changes made to data.
EventDispatcher class	Let you add and remove event listeners so that your code can react to events appropriately.
Iterator interface (Flash Professional only)	Lets you step through the objects that a collection contains.
MenuDataProvider class	Lets XML instances assigned to a `Menu.dataProvider` property use methods and properties to manipulate their own data as well as the associated menu views.
RectBorder class	Describes the styles used to control component borders.
SimpleButton class	Lets you control some aspects of button appearance and behavior.
TransferObject interface	Defines a set of methods that items managed by the DataSet component must implement.

Item	Description
TreeDataProvider interface (Flash Professional only)	A set of properties and methods used to create XML for the `Tree.dataProvider` property.
UIComponent class	Provides methods, properties, and events that allow components to share some common behavior.
UIEventDispatcher class	Allows components to emit certain events. This class is mixed in to the UIComponent class.
UIObject class	The base class for all version 2 components.

Accordion component (Flash Professional only)

The Accordion component is a navigator that contains a sequence of children that it displays one at a time. The children must be objects that inherit from the UIObject class (which includes all components and screens built with version 2 of the Macromedia Component Architecture); most often, children are a subclass of the View class. This includes movie clips assigned to the class mx.core.View. To maintain tabbing order in an accordion's children, the children must also be instances of the View class.

An accordion creates and manages header buttons that a user can click to navigate between the accordion's children. An accordion has a vertical layout with header buttons that span the width of the component. One header is associated with each child, and each header belongs to the accordion—not to the child. When a user clicks a header, the associated child is displayed below that header. The transition to the new child uses a transition animation.

An accordion with children accepts focus, and changes the appearance of its headers to display focus. When a user tabs into an accordion, the selected header displays the focus indicator. An accordion with no children does not accept focus. Clicking components that can take focus within the selected child gives them focus. When an Accordion instance has focus, you can use the following keys to control it:

Key	Description
Down Arrow, Right Arrow	Moves focus to the next child header. Focus cycles from last to first without changing the selected child.
Up Arrow, Left Arrow	Moves focus to the previous child header. Focus cycles from first to last without changing the selected child.
End	Selects the last child.
Enter/Space	Selects the child associated with the header that has focus.
Home	Selects the first child.
Page Down	Selects the next child. Selection cycles from the last child to the first child.
Page Up	Selects the previous child. Selection cycles from the first child to the last child.
Shift+Tab	Moves focus to the previous component. This component may be inside the selected child, or outside the accordion; it is never another header in the same accordion.
Tab	Moves focus to the next component. This component may be inside the selected child, or outside the accordion; it is never another header in the same accordion.

The Accordion component cannot be made accessible to screen readers.

Using the Accordion component (Flash Professional only)

You can use the Accordion component to present multipart forms. For example, a three-child accordion might present forms where the user fills out her shipping address, billing address, and payment information for an e-commerce transaction. Using an accordion instead of multiple web pages minimizes server traffic and allows the user to maintain a better sense of progress and context in an application.

Accordion parameters

You can set the following authoring parameters for each Accordion component instance in the Property inspector or in the Component inspector:

childSymbols is an array that specifies the linkage identifiers of the library symbols to be used to create the accordion's children. The default value is `[]` (an empty array).

childNames is an array that specifies the instance names of the accordion's children. The values you enter will be the instance names for the child symbols you specify in the childSymbols parameter. The default value is `[]` (an empty array).

childLabels is an array that specifies the text labels to use on the accordion's headers. The default value is `[]` (an empty array).

childIcons is an array that specifies the linkage identifiers of the library symbols to be used as the icons on the accordion's headers. The default value is `[]` (an empty array).

You can write ActionScript to control additional options for the Accordion component using its properties, methods, and events. For more information, see "Accordion class (Flash Professional only)" on page 105.

Creating an application with the Accordion component

In this example, an application developer is building the checkout section of an online store. The design calls for an accordion with three forms in which a user enters a shipping address, a billing address, and payment information. The shipping address and billing address forms are identical.

To use screens to add an Accordion component to an application:

1. In Flash, select File > New and select Flash Form Application.
2. Double-click the text Form1, and enter the name **addressForm**.

   Although it doesn't appear in the library, the addressForm screen is a symbol of the Screen class. Because the Screen class is a subclass of the View class, an accordion can use it as a child.
3. With the form selected, in the Property inspector, set the form's visible property to `false`.

   This hides the contents of the form in the application; the form only appears in the accordion.
4. Drag components such as Label and TextInput from the Components panel onto the form to create a mock address form; arrange them, and set their properties in the Parameters tab of the Component inspector.

   Position the form elements in the upper left corner of the form. This corner of the form is placed in the upper left corner of the accordion.

5. Repeat steps 2-4 to create a screen named **checkoutForm**.
6. Create a new screen named **accordionForm**.
7. Drag an Accordion component from the Components panel to the accordionForm form, and name it **myAccordion**.
8. With myAccordion selected, in the Property inspector, do the following:
   - For the childSymbols property, enter **addressForm**, **addressForm**, and **checkoutForm**.

     These strings specify the names of the screens used to create the accordion's children.

     Note: The first two children are instances of the same screen, because the shipping address form and the billing address form are identical.
   - For the childNames property, enter **shippingAddress**, **billingAddress**, and **checkout**.

     These strings are the ActionScript names of the accordion's children.
   - For the childLabels property, enter **Shipping Address**, **Billing Address**, and **Checkout**.

     These strings are the text labels on the accordion headers.
9. Select Control > Test Movie.

To add an Accordion component to an application:

1. Select File > New and create a new Flash document.
2. Select Insert > New Symbol and name it **AddressForm**.
3. In the Create New Symbol dialog box, click the Advanced button and select Export for ActionScript. In the AS 2.0 Class field, enter **mx.core.View**.

   To maintain tabbing order in an accordion's children, the children must also be instances of the View class.
4. Drag components such as Label and TextInput from the Components panel onto the Stage to create a mock address form; arrange them, and set their properties in the Parameters tab of the Component inspector.

   Position the form elements in relation to 0,0 (the middle) on the Stage. The 0,0 coordinate of the movie clip is placed in the upper left corner of the accordion.
5. Select Edit > Edit Document to return to the main Timeline.
6. Repeat steps 2-5 to create a movie clip named **CheckoutForm**.
7. Drag an Accordion component from the Components panel to add it to the Stage on the main Timeline.
8. In the Property inspector, do the following:
   - Enter the instance name **myAccordion**.
   - For the `childSymbols` property, enter **AddressForm**, **AddressForm**, and **CheckoutForm**.

     These strings specify the names of the movie clips used to create the accordion's children.

     Note: The first two children are instances of the same movie clip, because the shipping address form and the billing address form are identical.

- For the `childNames` property, enter **shippingAddress**, **billingAddress**, and **checkout**.

  These strings are the ActionScript names of the accordion's children.
- For the `childLabels` property, enter **Shipping Address**, **Billing Address**, and **Checkout**.

  These strings are the text labels on the accordion headers.
- For the `childIcons` property, enter **AddressIcon**, **AddressIcon**, and **CheckoutIcon**.

  These strings specify the linkage identifiers of the movie clip symbols that are used as the icons on the accordion headers. You must create these movie clip symbols if you want icons in the headers.

9. Select Control > Test Movie.

To use ActionScript to add children to an Accordion component:

1. Select File > New and create a Flash document.
2. Drag an Accordion component from the Components panel to the Stage.
3. In the Property inspector, enter the instance name **myAccordion**.
4. Drag a TextInput component to the Stage and delete it.

   This adds the component to the library so that you can dynamically instantiate it in step 6.
5. In the Actions panel on Frame 1 of the Timeline, enter the following:

```
myAccordion.createChild("View", "shippingAddress", {label: "Shipping
  Address"});
myAccordion.createChild("View", "billingAddress", {label: "Billing
  Address"});
myAccordion.createChild("View", "payment", {label: "Payment"});
```

   This code calls the `createChild()` method to create its child views.
6. In the Actions panel on Frame 1, below the code you entered in step 5, enter the following code:

```
var o = myAccordion.shippingAddress.createChild("TextInput", "firstName");
o.move(20, 38);
o.setSize(116, 20);
o = myAccordion.shippingAddress.createChild("TextInput", "lastName");
o.move(175, 38);
o.setSize(145, 20);
```

   This code adds component instances (two TextInput components) to the accordion's children.

Customizing the Accordion component (Flash Professional only)

You can transform an Accordion component horizontally and vertically during authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`).

The `setSize()` method and the Transform tool change only the width of the accordion's headers and the width and height of its content area. The height of the headers and the width and height of the children are not affected. Calling the `setSize()` method is the only way to change the bounding rectangle of an accordion.

If the headers are too small to contain their label text, the labels are clipped. If the content area of an accordion is smaller than a child, the child is clipped.

Using styles with the Accordion component

You can set style properties to change the appearance of the border and background of an Accordion component.

If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

An Accordion component uses the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. This is the only color style that doesn't inherit its value. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`.
`backgroundColor`	Both	The background color. The default color is white.
`border styles`	Both	The Accordion component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The Accordion component's default border style value is `"solid"`.
`headerHeight`	Both	The height of the header buttons, in pixels. The default value is 22.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for the header labels. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font of the header labels. The default value is 10.
`fontStyle`	Both	The font style for the header labels; either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight for the header labels; either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textDecoration`	Both	The text decoration; either `"none"` or `"underline"`.

Style	Theme	Description
openDuration	Both	The duration, in milliseconds, of the transition animation.
openEasing	Both	A reference to a tweening function that controls the animation. Defaults to sine in/out. For more information, see "Customizing component animations" on page 75.

Using skins with the Accordion component

The Accordion component uses skins to represent the visual states of its header buttons. To skin the buttons and title bar while authoring, modify skin symbols in the Flash UI Components 2/Themes/MMDefault/Accordion Assets skins states folder in the library of one of the themes FLA files. For more information, see "About skinning components" on page 80.

An Accordion component is composed of its border, background, header buttons, and children. The border and background are provided by the RectBorder class by default. For information on skinning the RectBorder class, see "RectBorder class" on page 647. You can sking the headers with the skins listed below.

Property	Description	Default value
falseUpSkin	The up (normal) state of the header above all collapsed children.	accordionHeaderSkin
falseDownSkin	The pressed state of the header above all collapsed children.	accordionHeaderSkin
falseOverSkin	The rolled-over state of the header above all collapsed children.	accordionHeaderSkin
falseDisabled	The disabled state of the header above all collapsed children.	accordionHeaderSkin
trueUpSkin	The up (normal) state of the header above the expanded child.	accordionHeaderSkin
trueDownSkin	The pressed state of the header above the expanded child.	accordionHeaderSkin
trueOverSkin	The rolled-over state of the header above the expanded child.	accordionHeaderSkin
trueDisabledSkin	The disabled state of the header above the expanded child.	accordionHeaderSkin

Using ActionScript to draw the Accordion header

The default headers in both the Halo and Sample themes use the same skin element for all states and draw the actual graphics through ActionScript. The Halo implementation uses an extension of the RectBorder class and custom drawing API code to draw the states. The Sample implementation uses the same skin and the same ActionScript class as the Button skin.

To create an ActionScript class to use as the skin and provide different states, the skin can read the `borderStyle` style property of the skin to determine the state. The following table shows the border style that is set for each skin:

Property	Border style
`falseUpSkin`	`falseup`
`falseDownSkin`	`falsedown`
`falseOverSkin`	`falserollover`
`falseDisabled`	`falsedisabled`
`trueUpSkin`	`trueup`
`trueDownSkin`	`truedown`
`trueOverSkin`	`truerollover`
`trueDisabledSkin`	`truedisabled`

To create an ActionScript customized Accordion header skin:

1. Create a new ActionScript class file.

   For this example, name the file **RedGreenBlueHeader.as**.

2. Copy the following ActionScript to the file:

```
import mx.skins.RectBorder;
import mx.core.ext.UIObjectExtensions;

class RedGreenBlueHeader extends RectBorder
{
  static var symbolName:String = "RedGreenBlueHeader";
  static var symbolOwner:Object = RedGreenBlueHeader;

  function size():Void
  {
    var c:Number; // color
    var borderStyle:String = getStyle("borderStyle");

    switch (borderStyle) {
      case "falseup":
      case "falserollover":
      case "falsedisabled":
        c = 0x7777FF;
        break;
      case "falsedown":
        c = 0x77FF77;
        break;
      case "trueup":
      case "truedown":
      case "truerollover":
      case "truedisabled":
        c = 0xFF7777;
        break;
    }
```

```
        clear();
        lineStyle(0, 0, 100);
        beginFill(c, 100);
        drawRect(0, 0, __width, __height);
        endFill();
    }

    // required for skins
    static function classConstruct():Boolean
    {
        UIObjectExtensions.Extensions();
        _global.skinRegistry["AccordionHeaderSkin"] = true;
        return true;
    }
    static var classConstructed:Boolean = classConstruct();
    static var UIObjectExtensionsDependency = UIObjectExtensions;
}
```

This class creates a square box based on the border style: a blue box for the false up, rollover, and disabled states; a green box for the normal pressed state; and a red box for the expanded child.

3. Save the file.
4. Create a new FLA file.
5. Save the FLA file in the same folder as the AS file.
6. Create a new symbol by selecting Insert > New Symbol.
7. Set the name to `AccordionHeaderSkin`.
8. If the advanced view is not displayed, click the Advanced button.
9. Select Export for ActionScript.

   The identifier will be automatically filled out with `AccordionHeaderSkin`.
10. Set the AS 2.0 class to `RedGreenBlueHeader`.
11. Ensure that Export in First Frame is already selected, and click OK.
12. Drag an Accordion component to the Stage.
13. Set the Accordion properties so that they display several children.

    For example, set the `childLabels` to an array of `[One,Two,Three]` and `childNames` to an array of `[one,two,three]`.
14. Select Control > Test Movie.

Using movie clips to customize the Accordion header skin

The above example demonstrates how to use an ActionScript class to customize the Accordion header skin, which is the method used by the skins provided in both the Halo and Sample themes. However, because the example uses simple colored boxes, it is simpler in this case to use different movie clip symbols as header skins.

To create movie clip symbols for Accordion header skins:

1. Create a new FLA file.
2. Create a new symbol by selecting Insert > New Symbol.
3. Set the name to `RedAccordionHeaderSkin`.
4. If the advanced view is not displayed, click the Advanced button.
5. Select Export for ActionScript.

   The identifier will be automatically filled out with `RedAccordionHeaderSkin`.
6. Leave the AS 2.0 Class text box blank.
7. Ensure that Export in First Frame is already selected, and click OK.
8. Open the new symbol for editing.
9. Use the drawing tools to create a box with a red fill and black line.
10. Set the border style to hairline.
11. Set the box, including the border, so that it is positioned at (0,0) and has a width and height of 100.

    The ActionScript code will size the skin as needed.
12. Repeat steps 2-11 and create green and blue skins, named accordingly.
13. Click the Back button to return to the main Timeline.
14. Drag an Accordion component to the stage.
15. Set the Accordion properties so that they display several children.

    For example, set `childLabels` to an array of `[One,Two,Three]` and `childNames` to an array of `[one,two,three]`.
16. Copy the following ActionScript code to the Actions panel with the Accordion instance selected:

```
onClipEvent(initialize) {
  falseUpSkin = "RedAccordionHeaderSkin";
  falseDownSkin = "GreenAccordionHeaderSkin";
  falseOverSkin = "RedAccordionHeaderSkin";
  falseDisabled = "RedAccordionHeaderSkin";
  trueUpSkin = "BlueAccordionHeaderSkin";
  trueDownSkin = "BlueAccordionHeaderSkin";
  trueOverSkin = "BlueAccordionHeaderSkin";
  trueDisabledSkin = "BlueAccordionHeaderSkin";
}
```

17. Select Control > Test Movie.

Accordion class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > Accordion

ActionScript Class Name mx.containers.Accordion

An Accordion component contains children that are displayed one at a time. Each child has a corresponding header button that is created when the child is created. A child must be an instance of UIObject.

A movie clip symbol automatically becomes an instance of the UIObject class when it becomes a child of an accordion. However, to maintain tabbing order in an accordion's children, the children must also be instances of the View class. If you use a movie clip symbol as a child, set its AS 2.0 Class field to mx.core.View so that it inherits from the View class.

Setting a property of the Accordion class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

Each component class has a `version` property that is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.containers.Accordion.version);
```

Note: The code `trace(myAccordionInstance.version);` returns `undefined`.

Method summary for the Accordion class

The following table lists methods of the Accordion class.

Method	Description
`Accordion.createChild()`	Creates a child for an Accordion instance.
`Accordion.createSegment()`	Creates a child for an Accordion instance. The parameters for this method are different from those of the `createChild()` method.
`Accordion.destroyChildAt()`	Destroys a child at a specified index position.
`Accordion.getChildAt()`	Gets a reference to a child at a specified index position.

Methods inherited from the UIObject class

The following table lists the methods the Accordion class inherits from the UIObject class. When calling these methods from the Accordion object, use the form *`accordionInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.

Method	Description
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from UIComponent class

The following table lists the methods the Accordion class inherits from the UIComponent class. When calling these methods from the Accordion object, use the form *accordionInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the Accordion class

The following table lists properties of the Accordion class.

Property	Description
Accordion.numChildren	The number of children of an Accordion instance.
Accordion.selectedChild	A reference to the selected child.
Accordion.selectedIndex	The index position of the selected child.

Properties inherited from the UIObject class

The following table lists the properties the Accordion class inherits from the UIObject class. When accessing these properties, use the form *accordionInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.

Property	Description
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Accordion class inherits from the UIComponent class. When accessing these properties, use the form *accordionInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the Accordion class

The following table lists an event of the Accordion class.

Event	Description
Accordion.change	Broadcast to all registered listeners when the `selectedIndex` and `selectedChild` properties of an accordion change because of a user's mouse click or keypress.

Events inherited from the UIObject class

The following table lists the events the Accordion class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Accordion class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

Accordion.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  // insert your code here
}
myAccordionInstance.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners when the `selectedIndex` and `selectedChild` properties of an accordion change. This event is broadcast only when a user's mouse click or keypress changes the value of `selectedChild` or `selectedIndex`—not when the value is changed with ActionScript. This event is broadcast before the transition animation occurs.

Version 2 components use a dispatcher/listener event model. The Accordion component dispatches a `change` event when one of its buttons is clicked and the event is handled by a function (also called a *handler*) on a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it a reference to the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

The Accordion change event also contains two unique event object properties:

- `newValue` Number; the index of the child that is about to be selected.
- `prevValue` Number; the index of the child that was previously selected.

Example

In the following example, a handler called `myAccordionListener` is defined and passed to the `myAccordion.addEventListener()` method as the second parameter. The event object is captured by the `change` handler in the *`eventObject`* parameter. When the `change` event is broadcast, a `trace` statement is sent to the Output panel.

```
myAccordionListener = new Object();
myAccordionListener.change = function(){
  trace("Changed to different view");
}
myAccordion.addEventListener("change", myAccordionListener);
```

Accordion.createChild()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.createChild(classOrSymbolName, instanceName[, initialProperties])
```

Parameters

`classOrSymbolName` Either the constructor function for the class of the UIObject to be instantiated, or the linkage name (a reference to the symbol to be instantiated). The class must be UIObject or a subclass of UIObject, but most often it is View object or a subclass of View.

`instanceName` The instance name of the new instance.

`initialProperties` An optional parameter that specifies initial properties for the new instance. You can use the following properties:

- `label` A string that specifies the text label that the new child instance uses on its header.
- `icon` A string that specifies the linkage identifier of the library symbol that the child uses for the icon on its header.

Returns

A reference to an instance of the UIObject that is the newly created child.

Description

Method (inherited from View); creates a child for the accordion. The newly created child is added to the end of the list of children owned by the accordion. Use this method to place views inside the accordion. The created child is an instance of the class or movie clip symbol specified in the *`classOrSymbolName`* parameter. You can use the `label` and `icon` properties to specify a text label and an icon for the associated accordion header for each child in the *`initialProperties`* parameter.

When each child is created, it is assigned an index number in the order of creation and the `numChildren` property is increased by 1.

Example

The following code creates an instance of the `PaymentForm` movie clip symbol named `payment` as the last child of `myAccordion`:

```
var child = myAccordion.createChild("PaymentForm", "payment", {label:
  "Payment", Icon: "payIcon"});
child.cardType.text = "Visa";
child.cardNumber.text = "1234567887654321";
```

The following code creates a child that is an instance of the View class:

```
var child = myAccordion.createChild(mx.core.View, "payment", {label:
  "Payment", Icon: "payIcon"});
child.cardType.text = "Visa";
child.cardNumber.text = "1234567887654321";
```

The following code also creates a child that is an instance of the View class, but it uses `import` to reference the constructor for the View class:

```
import mx.core.View
var child = myAccordion.createChild(View, "payment", {label: "Payment", Icon:
  "payIcon"});
child.cardType.text = "Visa";
child.cardNumber.text = "1234567887654321";
```

Accordion.createSegment()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.createSegment(classOrSymbolName, instanceName[, label[, icon]])
```

Parameters

`classOrSymbolName` Either a reference to the constructor function for the class of the UIObject to be instantiated, or the linkage name of the symbol to be instantiated. The class must be UIObject or a subclass of UIObject, but most often it is View or a subclass of View.

`instanceName` The instance name of the new instance.

`label` A string that specifies the text label that the new child instance uses on its header. This parameter is optional.

`icon` A string reference to the linkage identifier of the library symbol that the child uses for the icon on its header. This parameter is optional.

Returns

A reference to the newly created UIObject instance.

Description

Method; creates a child for the accordion. The newly created child is added to the end of the list of children owned by the accordion. Use this method to place views inside the accordion. The created child is an instance of the class or movie clip symbol specified in the *classOrSymbolName* parameter. You can use the *label* and *icon* parameters to specify a text label and an icon for the associated accordion header for each child.

The `createSegment()` method differs from the `createChild()` method in that *label* and *icon* are passed directly as parameters, not as properties of an *initalProperties* parameter.

When each child is created, it is assigned an index number in the order of creation, and the `numChildren` property is increased by 1.

Example

The following example creates an instance of the `PaymentForm` movie clip symbol named `payment` as the last child of `myAccordion`:

```
var child = myAccordion.createSegment("PaymentForm", "payment", "Payment",
  "payIcon");
child.cardType.text = "Visa";
child.cardNumber.text = "1234567887654321";
```

The following code creates a child that is an instance of the View class:

```
var child = myAccordion.createSegment(mx.core.View, "payment", {label:
  "Payment", Icon: "payIcon"});
child.cardType.text = "Visa";
child.cardNumber.text = "1234567887654321";
```

The following code also creates a child that is an instance of the View class, but it uses `import` to reference the constructor for the View class:

```
import mx.core.View
var child = myAccordion.createSegment(View, "payment", {label: "Payment",
  Icon: "payIcon"});
child.cardType.text = "Visa";
child.cardNumber.text = "1234567887654321";
```

Accordion.destroyChildAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.destroyChildAt(index)
```

Parameters

index The index number of the accordion child to destroy. Each child of an accordion is assigned a zero-based index number in the order in which it was created.

Returns

Nothing.

Description

Method (inherited from View); destroys one of the accordion's children. The child to be destroyed is specified by its index, which is passed to the method in the *index* parameter. Calling this method destroys the corresponding header as well.

If the destroyed child is selected, a new selected child is chosen. If there is a next child, it is selected. If there is no next child, the previous child is selected. If there is no previous child, the selection is undefined.

Note: Calling `destroyChildAt()` decreases the `numChildren` property by 1.

Example

The following code destroys the last child of `myAccordion`:

```
myAccordion.destroyChildAt(myAccordion.numChildren - 1);
```

See also

`Accordion.createChild()`

Accordion.getChildAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.getChildAt(index)
```

Parameters

index The index number of an accordion child. Each child of an accordion is assigned a zero-based index in the order in which it was created.

Returns

A reference to the instance of the UIObject at the specified index.

Description

Method; returns a reference to the child at the specified index. Each accordion child is given an index number for its position. This index number is zero-based, so the first child is 0, the second child is 1, and so on.

Example

The following code gets a reference to the last child of `myAccordion`:

```
var lastChild:UIObject = myAccordion.getChildAt(myAccordion.numChildren - 1);
```

Accordion.numChildren

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.numChildren
```

Description

Property (inherited from View); indicates the number of children (of type UIObject) in an Accordion instance. Headers are not counted as children.

Each accordion child is given an index number for its position. This index number is zero-based, so the first child is 0, the second child is 1, and so on. The code `myAccordion.numChild - 1` always refers to the last child added to an accordion. For example, if there were seven children in an accordion, the last child would have the index 6. The `numChildren` property is not zero-based, so the value of `myAccordion.numChildren` would be 7. The result of 7 - 1 is 6, which is the index number of the last child.

Example

The following example selects the last child:

```
myAccordion.selectedIndex = myAccordion.numChildren - 1;
```

Accordion.selectedChild

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.selectedChild
```

Description

Property; the selected child (of type UIObject) if one or more children exist; `undefined` if no children exist.

If the accordion has children, the code `myAccordion.selectedChild` is equivalent to the code `myAccordion.getChildAt(myAccordion.selectedIndex)`.

Setting this property to a child causes the accordion to begin the transition animation to display the specified child.

Changing the value of `selectedChild` also changes the value of `selectedIndex`.

The default value is `myAccordion.getChildAt(0)` if the accordion has children. If the accordion doesn't have children, the default value is `undefined`.

Example

The following example retrieves the label of the selected child view:

```
var selectedLabel = myAccordion.selectedChild.label;
```

The following example sets the payment form to be the selected child view:

```
myAccordion.selectedChild = myAccordion.payment;
```

See also

`Accordion.selectedIndex`

Accordion.selectedIndex

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.selectedIndex
```

Description

Property; the zero-based index of the selected child in an accordion with one or more children. For an accordion with no child views, the only valid value is `undefined`.

Each accordion child is given an index number for its position. This index number is zero-based, so the first child is 0, the second child is 1, and so on. The valid values of `selectedIndex` are 0, 1, 2, ... , *n* - 1, where *n* is the number of children.

Setting this property to a child causes the accordion to begin the transition animation to display the specified child.

Changing the value of `selectedIndex` also changes the value of `selectedChild`.

Example

The following example remembers the index of the selected child:

```
var oldSelectedIndex = myAccordion.selectedIndex;
```

The following example selects the last child:

```
myAccordion.selectedIndex = myAccordion.numChildren - 1;
```

See also

`Accordion.numChildren`, `Accordion.selectedChild`

Alert component (Flash Professional only)

The Alert component lets you display a window that presents the user with a message and response buttons. The window has a title bar that you can fill with text, a message that you can customize, and buttons whose labels you can change. An Alert window can have any combination of Yes, No, OK, and Cancel buttons, and you can change the button labels by using the `Alert.yesLabel`, `Alert.click`, `Alert.okLabel`, and `Alert.cancelLabel` properties. You cannot change the order of the buttons in an Alert window; the button order is always OK, Yes, No, Cancel. An Alert window closes when a user clicks any of its buttons.

To display an Alert window, call the `Alert.show()` method. In order to call the method successfully, the Alert component must be in the library. By dragging the Alert component from the Components panel to the Stage and then deleting the component, you add the component to the library without making it visible in the document.

The live preview for the Alert component is an empty window.

When you add an Alert component to an application, you can use the Accessibility panel to make the component's text and buttons accessible to screen readers. First, add the following line of code to enable accessibility:

```
mx.accessibility.AlertAccImpl.enableAccessibility();
```

Note: You enable accessibility for a component only once, regardless of how many instances you have of the component.

Using the Alert component (Flash Professional only)

You can use an Alert component whenever you want to announce something to a user. For example, you could display an alert when a user doesn't fill out a form properly, when a stock hits a certain price, or when a user quits an application without saving the session.

Alert parameters

The Alert component has no authoring parameters. You must call the ActionScript `Alert.show()` method to display an Alert window. You can use other ActionScript properties to modify the Alert window in an application. For more information, see "Alert class (Flash Professional only)" on page 119.

Creating an application with the Alert component

The following procedure explains how to add an Alert component to an application while authoring. In this example, the Alert component appears when a stock hits a certain price.

To create an application with the Alert component:

1. Double-click the Alert component in the Components panel to add it to the Stage.
2. Press Backspace (Windows) or Delete (Macintosh) to delete the component from the Stage.

   This adds the component to the library, but doesn't make it visible in the application.

3. In the Actions panel, enter the following code on Frame 1 of the Timeline to define an event handler for the `click` event:

```
import mx.controls.Alert;
myClickHandler = function (evt){
  if (evt.detail == Alert.OK){
    trace("start stock app");
    // startStockApplication();
  }
}
Alert.show("Launch Stock Application?", "Stock Price Alert", Alert.OK |
  Alert.CANCEL, this, myClickHandler, "stockIcon", Alert.OK);
```

This code creates an Alert window with OK and Cancel buttons. When the user clicks either button, Flash calls the `myClickHandler` function. But when the user clicks the OK button, Flash calls the `startStockApplication()` function.

Note: The `Alert.show()` method includes an optional parameter that displays an icon in the Alert window (in this example, an icon with the linkage identifier "stockIcon"). To include this icon in your test example, create a symbol named stockIcon and set it to Export for ActionScript in the Linkage Properties dialog box or the Create New Symbol dialog box.

4. Select Control > Test Movie.

Customizing the Alert component (Flash Professional only)

The Alert component positions itself in the center of the component that was passed as its *`parent`* parameter. The parent must be a UIComponent object. If it is a movie clip, you can register the clip as mx.core.View so that it inherits from UIComponent.

The Alert window automatically stretches horizontally to fit the message text or any buttons that are displayed. If you want to display large amounts of text, include line breaks in the text.

The Alert component does not respond to the `setSize()` method.

Using styles with the Alert component

You can set style properties to change the appearance of an Alert component. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

An Alert component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`backgroundColor`	Both	The background color. The default color is white for the Halo theme and 0xEFEBEF (light gray) for the Sample theme.

Style	Theme	Description
border styles	Both	The Alert component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The Alert component has a component-specific `borderStyle` setting of "`alert`" with the Halo theme and "`outset`" with the Sample theme.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textAlign`	Both	The text alignment: either `"left"`, `"right"`, or `"center"`. The default value is `"left"`.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.
`textIndent`	Both	A number indicating the text indent. The default value is 0.

The Alert component includes three different categories of text. Setting the text properties for the Alert component itself provides default values for all three categories, as shown here:

```
import mx.controls.Alert;
_global.styles.Alert.setStyle("color", 0x000099);
Alert.show("This is a test alert", "Title");
```

To set the text styles for one category individually, the Alert component provides static properties that are references to a CSSStyleDeclaration instance.

Static property	Text affected
`buttonStyleDeclaration`	Button
`messageStyleDeclaration`	Message
`titleStyleDeclaration`	Title

The following example demonstrates how to set the title of an Alert component to be italicized:

```
import mx.controls.Alert;
import mx.styles.CSSStyleDeclaration;

var titleStyles = new CSSStyleDeclaration();
titleStyles.setStyle("fontWeight", "bold");
titleStyles.setStyle("fontStyle", "italic");

Alert.titleStyleDeclaration = titleStyles;

Alert.show("Name is a required field", "Validation Error");
```

The default title style declarations set `fontWeight` to `"bold"`. When you override the `titleStyleDeclaration` property, this default is also overridden, so you must explicitly set `fontWeight` to `"bold"` if that setting is desired.

Note: Text styles set on an Alert component provide default text styles to its components through style inheritance. For more information, see "Setting inheriting styles on a container" on page 71.

Using skins with the Alert component

The Alert component extends the Window component and uses its title background skin for the title background, a RectBorder class instance for its border, and Button skins for the visual states of its buttons. To skin the buttons and title bar while authoring, modify the Flash UI Components 2/Themes/MMDefault/Window Assets/Elements/TitleBackground and Flash UI Components 2/Themes/MMDefault/Button Assets/ButtonSkin symbols. For more information, see "About skinning components" on page 80. The border and background are provided by the RectBorder class by default. For information on skinning the RectBorder class, see "RectBorder class" on page 647.

An Alert component uses the following skin properties to dynamically skin the buttons and title bar:

Property	Description	Default value
`buttonUp`	The up state of the buttons.	`ButtonSkin`
`buttonUpEmphasized`	The up state of the default button.	`ButtonSkin`
`buttonDown`	The pressed state of the buttons.	`ButtonSkin`
`buttonDownEmphasized`	The pressed state of the default button.	`ButtonSkin`
`buttonOver`	The rolled-over state of the buttons.	`ButtonSkin`
`buttonOverEmphasized`	The rolled-over state of the default button.	`ButtonSkin`
`titleBackground`	The window title bar.	`TitleBackground`

To set the title of an Alert component to a custom movie clip symbol:

1. Create a new FLA file.
2. Create a new symbol by selecting Insert > New Symbol.
3. Set the name to `TitleBackground`.

4. If the advanced view is not displayed, click the Advanced button.
5. Select Export for ActionScript.
6. The identifier will be automatically filled out with `TitleBackground`.
7. Set the AS 2.0 class to `mx.skins.SkinElement`.

   SkinElement is a simple class that can be used for all skin elements that don't provide their own ActionScript impelmentation. It provides movement and sizing functionality required by the version 2 component framework.
8. Ensure that Export in First Frame is already selected.
9. Click OK.
10. Open the new symbol for editing.
11. Use the drawing tools to create a box with a red fill and black line.
12. Set the border style to hairline.
13. Set the box, including the border, so that is positioned at (0,0) and has a width of 100 and height of 22.

    The Alert component will set the proper width of the skin as needed, but it will use the existing height as the height of the title.
14. Click the Back button to return to the main Timeline.
15. Drag an Alert component to the Stage and delete it.

    This will add the Alert component to the library and available at run-time.
16. Add ActionScript code to the main Timeline to create a sample Alert instance.

```
import mx.controls.Alert;
Alert.show("This is a skinned Alert component","Title");
```

17. Select Control > Test Movie.

Alert class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > Window component > Alert

ActionScript Class Name mx.controls.Alert

To use the Alert component, you drag an Alert component to the Stage and delete it so that the component is in the document library but not visible in the application. Then you call `Alert.show()` to display an Alert window. You can pass parameters to `Alert.show()` that add a message, a title bar, and buttons to the Alert window.

Because ActionScript is asynchronous, the Alert component is not blocking, which means that the lines of ActionScript code that follow the call to `Alert.show()` run immediately. You must add listeners to handle the `click` events that are broadcast when a user clicks a button and then continue your code after the event is broadcast.

Note: In operating environments that are blocking (for example, Microsoft Windows), a call to `Alert.show()` does not return until the user has taken an action, such as clicking a button.

To understand more about the Alert class, see "Window component" on page 878 and "PopUpManager class" on page 601.

Method summary for the Alert class

The following table lists the method of the Alert class.

Method	Description
`Alert.show()`	Creates an Alert window with optional parameters.

Methods inherited from the UIObject class

The following table lists the methods the Alert class inherits from the UIObject class.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Alert class inherits from the UIComponent class.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Methods inherited from the Window class

The following table lists the methods the Alert class inherits from the Window class.

Method	Description
`Window.deletePopUp()`	Removes a window instance created by `PopUpManager.createPopUp()`.

Property summary for the Alert class

The following table lists properties of the Alert class.

Property	Description
`Alert.buttonHeight`	The height of each button, in pixels. The default value is 22.
`Alert.buttonWidth`	The width of each button, in pixels. The default value is 100.
`Alert.CANCEL`	A constant hexadecimal value indicating whether a Cancel button should be displayed in the Alert window.
`Alert.cancelLabel`	The label text for the Cancel button.
`Alert.click`	The label text for the No button.
`Alert.NO`	A constant hexadecimal value indicating whether a No button should be displayed in the Alert window.
`Alert.OK`	A constant hexadecimal value indicating whether an OK button should be displayed in the Alert window.
`Alert.okLabel`	The label text for the OK button.
`Alert.YES`	A constant hexadecimal value indicating whether a Yes button should be displayed in the Alert window.
`Alert.yesLabel`	The label text for the Yes button.

Properties inherited from the UIObject class

The following table lists the properties the Alert class inherits from the UIObject class. When calling these properties from the Alert object, use the form `Alert.`*`propertyName`*.

Property	Description
`UIObject.bottom`	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
`UIObject.height`	The height of the object, in pixels. Read-only.
`UIObject.left`	The left edge of the object, in pixels. Read-only.
`UIObject.right`	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
`UIObject.scaleX`	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
`UIObject.scaleY`	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
`UIObject.top`	The position of the top edge of the object, relative to its parent. Read-only.
`UIObject.visible`	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
`UIObject.width`	The width of the object, in pixels. Read-only.

Property	Description
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Alert class inherits from the UIComponent class. When calling these properties from the Alert object, use the form `Alert.propertyName`.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the Window class

The following table lists the properties the Alert class inherits from the Window class.

Property	Description
Window.closeButton	Indicates whether a close button is (`true`) or is not (`false`) included on the title bar.
Window.content	A reference to the content (root movie clip) of the window.
Window.contentPath	Sets the name of the content to display in the window.
Window.title	The text that appears in the title bar.
Window.titleStyleDeclaration	The style declaration that formats the text in the title bar.

Event summary for the Alert class

The following table lists an event of the Alert class.

Event	Description
Alert.click	Broadcast when a button in an Alert window is clicked.

Events inherited from the UIObject class

The following table lists the events the Alert class inherits from the UIObject class. When calling these events from the Alert object, use the form `Alert.eventName`.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.

Event	Description
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Alert class inherits from the UIComponent class. When calling these events from the Alert object, use the form `Alert.eventName`.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Events inherited from the Window class

The following table lists the events the Alert class inherits from the Window class.

Event	Description
Window.click	Broadcast when the close button is clicked (released).
Window.complete	Broadcast when a window is created.
Window.mouseDownOutside	Broadcast when the mouse is clicked (released) outside the modal window.

Alert.buttonHeight

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.buttonHeight
```

Description

Property (class); a class (static) property that changes the height of the buttons. The default value is 22.

See also

Alert.buttonWidth

Alert.buttonWidth

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.buttonWidth
```

Description

Property (class); a class (static) property that changes the width of the buttons. The default value is 100.

See also

`Alert.buttonHeight`

Alert.CANCEL

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.CANCEL
```

Description

Property (constant); a property with the constant hexadecimal value 0x8. This property can be used for the *flags* or *defaultButton* parameter of the `Alert.show()` method. When used as a value for the *flags* parameter, this property indicates that a Cancel button should be displayed in the Alert window. When used as a value for the *defaultButton* parameter, the Cancel button has initial focus and is triggered when the user presses Enter (Windows) or Return (Macintosh). If the user tabs to another button, that button is triggered when the user presses Enter.

Example

The following example uses `Alert.CANCEL` and `Alert.OK` as values for the *flags* parameter and displays an Alert component with an OK button and a Cancel button:

```
import mx.controls.Alert;
Alert.show("This is a generic Alert window", "Alert Test", Alert.OK |
  Alert.CANCEL, this);
```

Alert.cancelLabel

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.cancelLabel
```

Description

Property (class); a class (static) property that indicates the label text on the Cancel button.

Example

The following example sets the Cancel button's label to "cancellation":

```
Alert.cancelLabel = "cancellation";
```

Alert.click

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
clickHandler = function(eventObject){
  // insert code here
}
Alert.show(message[, title[, flags[, parent[, clickHandler[, icon[,
  defaultButton]]]]]])
```

Description

Event; broadcast to the registered listener when the OK, Yes, No, or Cancel button is clicked.

Version 2 components use a dispatcher/listener event model. The Alert component dispatches a `click` event when one of its buttons is clicked and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You call the `Alert.show()` method and pass it the name of the handler as a parameter. When a button in the Alert window is clicked, the listener is called.

When the event occurs, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Alert.click` event's event object has an additional `detail` property whose value is `Alert.OK`, `Alert.CANCEL`, `Alert.YES`, or `Alert.NO`, depending on which button was clicked. For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myClickHandler` is defined and passed to the `Alert.show()` method as the fifth parameter. The event object is captured by `myClickHandler` in the `evt` parameter. The `detail` property of the event object is then used in a `trace` statement to send the name of the button that was clicked (`Alert.OK` or `Alert.CANCEL`) to the Output panel.

```
import mx.controls.Alert;
myClickHandler = function(evt){
  if(evt.detail == Alert.OK){
    trace(Alert.okLabel);
  }else if (evt.detail == Alert.CANCEL){
    trace(Alert.cancelLabel);
  }
}
Alert.show("This is a test of errors", "Error", Alert.OK | Alert.CANCEL, this,
  myClickHandler);
```

Alert.NO

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.NO
```

Description

Property (constant); a property with the constant hexadecimal value 0x2. This property can be used for the *`flags`* or *`defaultButton`* parameter of the `Alert.show()` method. When used as a value for the *`flags`* parameter, this property indicates that a No button should be displayed in the Alert window. When used as a value for the *`defaultButton`* parameter, the Cancel button has initial focus and is triggered when the user presses Enter (Windows) or Return (Macintosh). If the user tabs to another button, that button is triggered when the user presses Enter.

Example

The following example uses `Alert.NO` and `Alert.YES` as values for the *`flags`* parameter and displays an Alert component with a No button and a Yes button:

```
import mx.controls.Alert;
Alert.show("This is a generic Alert window", "Alert Test", Alert.NO |
  Alert.YES, this);
```

Alert.noLabel

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.noLabel
```

Description

Property (class); a class (static) property that indicates the label text on the No button.

Example

The following example sets the No button's label to "nyet":

```
Alert.noLabel = "nyet";
```

Alert.OK

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.OK
```

Description

Property (constant); a property with the constant hexadecimal value 0x4. This property can be used for the *flags* or *defaultButton* parameter of the `Alert.show()` method. When used as a value for the *flags* parameter, this property indicates that an OK button should be displayed in the Alert window. When used as a value for the *defaultButton* parameter, the OK button has initial focus and is triggered when the user presses Enter (Windows) or Return (Macintosh). If the user tabs to another button, that button is triggered when the user presses Enter.

Example

The following example uses `Alert.OK` and `Alert.CANCEL` as values for the *flags* parameter and displays an Alert component with an OK button and a Cancel button:

```
import mx.controls.Alert;
Alert.show("This is a generic Alert window", "Alert Test", Alert.OK |
  Alert.CANCEL, this);
```

Alert.okLabel

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.okLabel
```

Description

Property (class); a class (static) property that indicates the label text on the OK button.

Example

The following example sets the OK button's label to "okay":

```
Alert.okLabel = "okay";
```

Alert.show()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.show(message[, title[, flags[, parent[, clickHandler[, icon[,
  defaultButton]]]]]])
```

Parameters

message The message to display.

title The text in the Alert title bar. This parameter is optional; if you omit it, the title bar is blank.

flags An optional parameter that indicates the buttons to display in the Alert window. The default value is `Alert.OK`, which displays an OK button. When you use more than one value, separate the values with a `|` character. Use one or more of the following values: `Alert.OK`, `Alert.CANCEL`, `Alert.YES`, `Alert.NO`.

You can also use `Alert.NONMODAL` to indicate that the Alert window is nonmodal. A nonmodal window allows a user to interact with other windows in the application.

parent The parent window for the Alert component. The Alert window centers itself in the parent window. Use the value `null` or `undefined` to specify the _root Timeline. The parent window must inherit from the UIComponent class. You can register the parent window with mx.core.View to cause it to inherit from UIComponent. This parameter is optional.

clickHandler A handler for the `click` events broadcast when the buttons are clicked. In addition to the standard click event object properties, there is an additional `detail` property, which contains the flag value of the button that was clicked (`Alert.OK`, `Alert.CANCEL`, `Alert.YES`, `Alert.NO`). This handler can be a function or an object. For more information, see "Using listeners to handle events" on page 56.

icon A string that is the linkage identifier of a symbol in the library; this symbol is used as an icon displayed to the left of the alert text. This parameter is optional.

defaultButton Indicates which button has initial focus and is clicked when a user presses Enter (Windows) or Return (Macintosh). If a user tabs to another button, that button is triggered when the Enter key is pressed.

This parameter can be one of the following values: `Alert.OK`, `Alert.CANCEL`, `Alert.YES`, `Alert.NO`.

Returns

The Alert instance that is created.

Description

Method (class); a class (static) method that displays an Alert window with a message, an optional title, optional buttons, and an optional icon. The title of the alert appears at the top of the window and is left-aligned. The icon appears to the left of the message text. The buttons are centered below the message text and the icon.

Example

The following code is a simple example of a modal Alert window with an OK button:

```
mx.controls.Alert.show("Hello, world!");
```

The following code defines a click handler that sends a message to the Output panel about which button was clicked:

```
import mx.controls.Alert;
myClickHandler = function(evt){
  trace ("button " + evt.detail + " was clicked");
}
Alert.show("This is a test of errors", "Error", Alert.OK | Alert.CANCEL, this,
  myClickHandler);
```

The event object's `detail` property returns a number to represent each button. The OK button is 4, the Cancel button is 8, the Yes button is 1, and the No button is 2.

Note: You must have an Alert component in the library for this code to display an alert. To add the component to the library, drag it to the Stage and then delete it.

Alert.YES

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.YES
```

Description

Property (constant); a property with the constant hexadecimal value 0x1. This property can be used for the *flags* or *defaultButton* parameter of the `Alert.show()` method. When used as a value for the *flags* parameter, this property indicates that a Yes button should be displayed in the Alert window. When used as a value for the *defaultButton* parameter, the Yes button has initial focus and is triggered when the user presses Enter (Windows) or Return (Macintosh). If the user tabs to another button, that button is triggered when the user presses Enter.

Example

The following example uses `Alert.NO` and `Alert.YES` as values for the *flags* parameter and displays an Alert component with a No button and a Yes button:

```
import mx.controls.Alert;
Alert.show("This is a generic Alert window", "Alert Test", Alert.NO |
  Alert.YES, this);
```

Alert.yesLabel

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Alert.yesLabel
```

Description

Property (class); a class (static) property that indicates the label text on the Yes button.

Example

The following example sets the OK button's label to "da":

```
Alert.yesLabel = "da";
```

Button component

The Button component is a resizable rectangular user interface button. You can add a custom icon to a button. You can also change the behavior of a button from push to toggle. A toggle button stays pressed when clicked and returns to its up state when clicked again.

A button can be enabled or disabled in an application. In the disabled state, a button doesn't receive mouse or keyboard input. An enabled button receives focus if you click it or tab to it. When a Button instance has focus, you can use the following keys to control it:

Key	Description
Shift+Tab	Moves focus to the previous object.
Spacebar	Presses or releases the component and triggers the `click` event.
Tab	Moves focus to the next object.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each Button instance reflects changes made to parameters in the Property inspector or Component inspector during authoring. However, in the live preview a custom icon is represented on the Stage by a gray square.

When you add the Button component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code:

```
mx.accessibility.ButtonAccImpl.enableAccessibility();
```

You enable accessibility for a component only once. regardless of how many instances you have of the component.

Using the Button component

A button is a fundamental part of any form or web application. You can use buttons wherever you want a user to initiate an event. For example, most forms have a Submit button. You could also add Previous and Next buttons to a presentation.

To add an icon to a button, you need to select or create a movie clip or graphic symbol to use as the icon. The symbol should be registered at 0,0 for appropriate layout on the button. Select the icon symbol in the Library panel, open the Linkage dialog box from the Library options menu, and enter a linkage identifier. This is the value to enter for the icon parameter in the Property inspector or Component inspector. You can also enter this value for the `Button.icon` ActionScript property.

Note: If an icon is larger than the button, it extends beyond the button's borders.

To designate a button as the default push button in an application (the button that receives the click event when a user presses Enter), use `FocusManager.defaultPushButton`.

Button parameters

You can set the following authoring parameters for each Button component instance in the Property inspector or in the Component inspector:

label sets the value of the text on the button; the default value is Button.

icon adds a custom icon to the button. The value is the linkage identifier of a movie clip or graphic symbol in the library; there is no default value.

toggle turns the button into a toggle switch. If `true`, the button remains in the down state when clicked and returns to the up state when clicked again. If `false`, the button behaves like a normal push button; the default value is `false`.

selected if the toggle parameter is `true`, this parameter specifies whether the button is pressed (`true`) or released (`false`). The default value is `false`.

labelPlacement orients the label text on the button in relation to the icon. This parameter can be one of four values: `left`, `right`, `top`, or `bottom`; the default value is `right`. For more information, see `Button.labelPlacement`.

You can write ActionScript to control these and additional options for the Button component using its properties, methods, and events. For more information, see "Button class" on page 139.

Creating an application with the Button component

The following procedure explains how to add a Button component to an application while authoring. In this example, the button is a Help button with a custom icon that opens a Help system when a user clicks it.

To create an application with the Button component:

1. Drag a Button component from the Components panel to the Stage.
2. In the Property inspector, enter the instance name **helpBtn**.
3. In the Property inspector, do the following:
   - Enter **Help** for the label parameter.
   - Enter **HelpIcon** for the icon parameter.

     To use an icon, there must be a movie clip or graphic symbol in the library with a linkage identifier to use as the icon parameter. In this example, the linkage identifier is HelpIcon.
   - Set the `toggle` property to `true`.
4. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
function click(evt){
  clippyHelper.enabled = evt.target.selected;
}
helpBtn.addEventListener("click", this);
```

   The last line of code adds a `click` event handler to the `helpBtn` instance. The handler enables and disables the `clippyHelper` instance, which could be a Help panel of some sort.

Customizing the Button component

You can transform a Button component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`) or any applicable properties and methods of the Button class (see "Button class" on page 139). Resizing the button does not change the size of the icon or label.

The bounding box of a Button instance is invisible and also designates the hit area for the instance. If you increase the size of the instance, you also increase the size of the hit area. If the bounding box is too small to fit the label, the label is clipped to fit.

If an icon is larger than the button, the icon extends beyond the button's borders.

Using styles with the Button component

You can set style properties to change the appearance of a button instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

A Button component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`backgroundColor`	Sample	The background color. The default value is 0xEFEBEF (light gray). The Halo theme uses 0xF8F8F8 (very light gray) for the button background color when the button is up and `themeColor` when the button is pressed. You can only modify the up background color in the Halo theme by skinning the button. See "Using skins with the Button component" on page 134.
`border styles`	Sample	The Button component uses a RectBorder instance as its border in the Sample theme and responds to the styles defined on that class. See "RectBorder class" on page 647. With the Halo theme, the Button component uses a custom rounded border whose colors cannot be modified except for `themeColor`.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).

Style	Theme	Description
embedFonts	Both	A Boolean value that indicates whether the font specified in fontFamily is an embedded font. This style must be set to true if fontFamily refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to true and fontFamily does not refer to an embedded font, no text will be displayed. The default value is false.
fontFamily	Both	The font name for text. The default value is "_sans".
fontSize	Both	The point size for the font. The default value is 10.
fontStyle	Both	The font style: either "normal" or "italic". The default value is "normal".
fontWeight	Both	The font weight: either "none" or "bold". The default value is "none". All components can also accept the value "normal" in place of "none" during a setStyle() call, but subsequent calls to getStyle() will return "none".
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".

Using skins with the Button component

The Button component includes 32 different skins that can be customized to correspond to the border and icon in 16 different states. To skin the Button component while authoring, create new movie clip symbols with the desired graphics and set the symbol linkage identifiers using ActionScript. (See "Using ActionScript to draw Button skins" on page 136.)

The default implementation of the Button skins provided with both the Halo and Sample themes uses the ActionScript drawing API to draw the button states, and uses a single movie clip symbol associated with one ActionScript class to provide all skins for the Button component.

The Button component has many skins because a button has so many states, and a border and icon for each state. The state of a Button instance is controlled by four properties and user interaction. The properties that affect skins include the following:

Property	Description
emphasized	Provides two different looks for Button instances and is typically used to highlight one button, such as the default button in a form.
enabled	Shows whether or not the button is allowing user interaction.
toggle	Toggle buttons provide a selected and unselected value and use different skins to demonstate the current value. For a Button instance whose toggle property is set to false, the false skins are used. When the toggle property is true, the skin depends on the selected property.
selected	When the toggle property is set to true, this property determines if the Button is selected (true or false). Different skins are used to identify the value and by default are the only way this value is depicted on screen.

If a button is enabled, it displays its over state when the pointer moves over it. The button receives input focus and displays its down state when it's pressed. The button returns to its over state when the mouse is released. If the pointer moves off the button while the mouse is pressed, the button returns to its original state and it retains input focus. If the toggle parameter is set to `true`, the state of the button does not change until the mouse is released over it.

If a button is disabled, it displays its disabled state, regardless of user interaction.

A Button component supports the following skin properties:

Property	Description
`falseUpSkin`	The up (normal) state.
`falseDownSkin`	The pressed state.
`falseOverSkin`	The over state.
`falseDisabledSkin`	The disabled state.
`trueUpSkin`	The toggled state.
`trueDownSkin`	The pressed-toggled state.
`trueOverSkin`	The over-toggled state.
`trueDisabledSkin`	The disabled-toggled state.
`falseUpSkinEmphasized`	The up (normal) state of an emphasized button.
`falseDownSkinEmphasized`	The pressed state of an emphasized button.
`falseOverSkinEmphasized`	The over state of an emphasized button.
`falseDisabledSkinEmphasized`	The disabled state of an emphasized button.
`trueUpSkinEmphasized`	The toggled state of an emphasized button.
`trueDownSkinEmphasized`	The pressed-toggled state of an emphasized button.
`trueOverSkinEmphasized`	The over-toggled state of an emphasized button.
`trueDisabledSkinEmphasized`	The disabled-toggled state of an emphasized button.
`falseUpIcon`	The icon up state.
`falseDownIcon`	The icon pressed state.
`falseOverIcon`	The icon over state.
`falseDisabledIcon`	The icon disabled state.
`trueUpIcon`	The icon toggled state.
`trueOverIcon`	The icon over-toggled state.
`trueDownIcon`	The icon pressed-toggled state.
`trueDisabledIcon`	The icon disabled-toggled state.
`falseUpIconEmphasized`	The icon up state of an emphasized button.
`falseDownIconEmphasized`	The icon pressed state of an emphasized button.
`falseOverIconEmphasized`	The icon over state of an emphasized button.

Property	Description
falseDisabledIconEmphasized	The icon disabled state of an emphasized button.
trueUpIconEmphasized	The icon toggled state of an emphasized button.
trueOverIconEmphasized	The icon over-toggled state of an emphasized button.
trueDownIconEmphasized	The icon pressed-toggled state of an emphasized button.
trueDisabledIconEmphasized	The icon disabled-toggled state of an emphasized button.

The default value for all skin properties ending in "Skin" is ButtonSkin, and the default for all "Icon" properties is undefined. The properties with the "Skin" suffix provide a background and border, whereas those with the "Icon" suffix provide a small icon.

In addition to the icon skins, the Button component also supports a standard icon property. The difference between the standard property and style property is that through the style property you can set icons for the individual states, whereas with the standard property only one icon can be set and it applies to all states. If a Button instance has both the icon property and icon style properties set, the instance may not behave as anticipated.

To see an interactive movie demonstrating when each skin is used, see Using Components Help.

Using ActionScript to draw Button skins

The default skins in both the Halo and Sample themes use the same skin element for all states and draw the actual graphics through ActionScript. The Halo implementation uses an extension of the RectBorder class and custom drawing API code to draw the states. The Sample implementation uses the same skin and the same ActionScript class as the Button skin.

To create an ActionScript class to use as the skin and provide different states, the skin can read the borderStyle style property of the skin and emphasized property of the parent to determine the state. The following table shows the border style that is set for each skin:

Property	Border style
falseUpSkin	falseup
falseDownSkin	falsedown
falseOverSkin	falserollover
falseDisabled	falsedisabled
trueUpSkin	trueup
trueDownSkin	truedown
trueOverSkin	truerollover
trueDisabledSkin	truedisabled

To create an ActionScript customized Button skin:

1. Create a new ActionScript class file.

   For this example, name the file RedGreenBlueSkin.as.

2. Copy the following ActionScript to the file:

```
import mx.skins.RectBorder;
import mx.core.ext.UIObjectExtensions;

class RedGreenBlueSkin extends RectBorder
{
  static var symbolName:String = "RedGreenBlueSkin";
  static var symbolOwner:Object = RedGreenBlueSkin;

  function size():Void
  {
    var c:Number; // color
    var borderStyle:String = getStyle("borderStyle");

    switch (borderStyle) {
      case "falseup":
      case "falserollover":
      case "falsedisabled":
        c = 0x7777FF;
        break;
      case "falsedown":
        c = 0x77FF77;
        break;
      case "trueup":
      case "truedown":
      case "truerollover":
      case "truedisabled":
        c = 0xFF7777;
        break;
    }

    clear();
    var thickness = _parent.emphasized ? 2 : 0;
    lineStyle(thickness, 0, 100);
    beginFill(c, 100);
    drawRect(0, 0, __width, __height);
    endFill();
  }

  // required for skins
  static function classConstruct():Boolean
  {
    UIObjectExtensions.Extensions();
    _global.skinRegistry["ButtonSkin"] = true;
    return true;
  }
  static var classConstructed:Boolean = classConstruct();
  static var UIObjectExtensionsDependency = UIObjectExtensions;
}
```

This class creates a square box based on the border style: a blue box for the false up, rollover, and disabled states; a green box for the normal pressed state; and a red box for the expanded child. It draws a hairline border in the normal case and a thick border if the button is emphasized.

3. Save the file.
4. Create a new FLA file.
5. Save the FLA file in the same folder as the AS file.
6. Create a new symbol by selecting Insert > New Symbol.
7. Set the name to `ButtonSkin`.
8. If the advanced view is not displayed, click the Advanced button.
9. Select Export for ActionScript.

   The identifier will be automatically filled out with `ButtonSkin`.
10. Set the AS 2.0 class to `RedGreenBlueSkin`.
11. Ensure that Export in First Frame is already selected, and click OK.
12. Drag a Button component to the Stage.
13. Select Control > Test Movie.

Using movie clips to customize Button skins

The above example demonstrates how to use an ActionScript class to customize the Button skin, which is the method used by the skins provided in both the Halo and Sample themes. However, because the example uses simple colored boxes, it is simpler in this case to use different movie clip symbols as the skins.

To create movie clip symbols for Button skins:

1. Create a new FLA file.
2. Create a new symbol by selecting Insert > New Symbol.
3. Set the name to `RedButtonSkin`.
4. If the advanced view is not displayed, click the Advanced button.
5. Select Export for ActionScript.

   The identifier will be automatically filled out with `RedButtonSkin`.
6. Set the AS 2.0 class to `mx.skins.SkinElement`.
7. Ensure that Export in First Frame is already selected, and click OK.
8. Open the new symbol for editing.
9. Use the drawing tools to create a box with a red fill and black line.
10. Set the border style to hairline.
11. Set the box, including the border, so that it is positioned at (0,0) and has a width and height of 100.

    The `SkinElement` class resizes the content as appropriate.

12. Repeat steps 2-11 and create green and blue skins, named accordingly.
13. Click the Back button to return to the main Timeline.
14. Drag a Button component to the Stage.
15. Set the `toggled` property value to `true` to see all three skins.
16. Copy the following ActionScript code to the Actions panel with the Button instance selected.

```
onClipEvent(initialize) {
  falseUpSkin = "BlueButtonSkin";
  falseDownSkin = "GreenButtonSkin";
  falseOverSkin = "BlueButtonSkin";
  falseDisabledSkin = "BlueButtonSkin";
  trueUpSkin = "RedButtonSkin";
  trueDownSkin = "RedButtonSkin";
  trueOverSkin = "RedButtonSkin";
  trueDisabledSkin = "RedButtonSkin";
}
```

17. Select Control > Test Movie.

Button class

Inheritance MovieClip > UIObject class > UIComponent class > SimpleButton class > Button

ActionScript Class Name mx.controls.Button

The properties of the Button class let you do the following at runtime: add an icon to a button, create a text label, and indicate whether the button acts as a push button or as a toggle switch.

Setting a property of the Button class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

The Button component uses the Focus Manager to override the default Flash Player focus rectangle and draw a custom focus rectangle with rounded corners. For more information, see "Creating custom focus navigation" on page 50.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.Button.version);
```

Note: The code `trace(myButtonInstance.version);` returns `undefined`.

The Button component class is different from the built-in ActionScript Button object.

Method summary for the Button class

There are no methods exclusive to the Button class.

Methods inherited from the UIObject class

The following table lists the methods the Button class inherits from the UIObject class. When calling these methods from the Button object, use the form *buttonInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Button class inherits from the UIComponent class. When calling these methods from the Button object, use the form *buttonInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the Button class

The following table lists properties of the Button class.

Property	Description
Button.icon	Specifies an icon for a button instance.
Button.label	Specifies the text that appears in a button.
Button.labelPlacement	Specifies the orientation of the label text in relation to an icon.

Properties inherited from the SimpleButton class

The following table lists the properties the Button class inherits from the SimpleButton class. When accessing these properties, use the form *buttonInstance.propertyName*.

Property	Description
SimpleButton.emphasized	Indicates whether a button has the look of a default push button.
SimpleButton.emphasizedStyleDeclaration	The style declaration when the emphasized property is set to true.
SimpleButton.selected	A Boolean value indicating whether the button is selected (true) or not (false). The default value is false.
SimpleButton.toggle	A Boolean value indicating whether the button behaves as a toggle switch (true) or not (false). The default value is false.

Properties inherited from the UIObject class

The following table lists the properties the Button class inherits from the UIObject class. When accessing these properties from the Button object, use the form *buttonInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Button class inherits from the UIComponent class. When accessing these properties from the Button object, use the form *buttonInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the Button class

There are no events exclusive to the Button class.

Events inherited from the SimpleButton class

The following table lists the events the Button class inherits from the SimpleButton class.

Property	Description
SimpleButton.click	Broadcast when a button is clicked.

Events inherited from the UIObject class

The following table lists the events the Button class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Button class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Button.icon

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

buttonInstance.icon

Description

Property; a string that specifies the linkage identifier of a symbol in the library to be used as an icon for a button instance. The icon can be a movie clip symbol or a graphic symbol with an upper left registration point. You must resize the button if the icon is too large to fit; neither the button nor the icon resizes automatically. If an icon is larger than a button, the icon extends over the borders of the button.

To create a custom icon, create a movie clip or graphic symbol. Select the symbol on the Stage in symbol-editing mode and enter 0 in both the X and Y boxes in the Property inspector. In the Library panel, select the movie clip and select Linkage from the Library options menu. Select Export for ActionScript, and enter an identifier in the Identifier text box.

The default value is an empty string (`""`), which indicates that there is no icon.

Use the `labelPlacement` property to set the position of the icon in relation to the button.

Note: The icon does not appear on the Stage in Flash. You must choose Control > Test Movie to see the icon.

Example

The following code assigns the movie clip from the Library panel with the linkage identifier `happiness` to the Button instance as an icon:

```
myButton.icon = "happiness"
```

See also

`Button.labelPlacement`

Button.label

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

buttonInstance.label

Description

Property; specifies the text label for a button instance. By default, the label appears centered on the button. Calling this method overrides the label authoring parameter specified in the Property inspector or the Component inspector. The default value is `"Button"`.

Example

The following code sets the label to "Remove from list":

```
buttonInstance.label = "Remove from list";
```

See also

`Button.labelPlacement`

Button.labelPlacement

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`buttonInstance``.labelPlacement`

Description

Property; sets the position of the label in relation to the icon. The default value is `"right"`. The following are the four possible values; the icon and label are always centered vertically and horizontally within the bounding area of the button:

- `"right"` The label is set to the right of the icon.
- `"left"` The label is set to the left of the icon.
- `"bottom"` The label is set below the icon.
- `"top"` The label is set above the icon.

Example

The following code sets the label to the left of the icon. The second line of the code sends the value of the `labelPlacement` property to the Output panel:

```
iconInstance.labelPlacement = "left";
trace(iconInstance.labelPlacement);
```

CellRenderer API

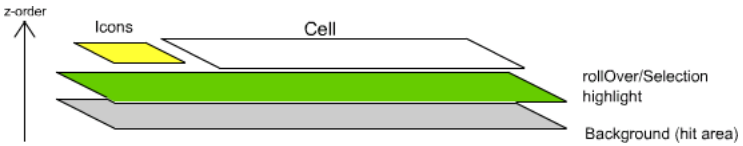
The CellRenderer API is a set of properties and methods that the list-based components (List, DataGrid, Tree, Menu, and ComboBox) use to manipulate and display custom cell content for each of their rows. This customized cell can contain a prebuilt component, such as a CheckBox component, or any class you create.

Understanding the List class

To use the CellRenderer API, you need an advanced understanding of the List class. The DataGrid, Tree, Menu, and ComboBox components are extensions of the List class, so understanding the List class lets you understand them as well.

About the composition of the List component

List components are composed of rows. These rows display rollover and selection highlights, are used as hit states for row selection, and play a vital part in scrolling. Aside from selection highlights and icons (such as the node icons and expander arrows of a Tree component), a row consists of one cell (or, in the case of the DataGrid component, many cells). In the default case, these cells are TextField objects that implement the CellRenderer API. However, you can tell a List component to use a different class of component as the cell for each row. The only requirement is that the class must implement the CellRenderer API, which the List component uses for communicating with the cell.



The stacking order of a row in a List or DataGrid component

Note: If a cell has button event handlers (`onPress` and so on), the background hit area may not receive input necessary to trigger the events.

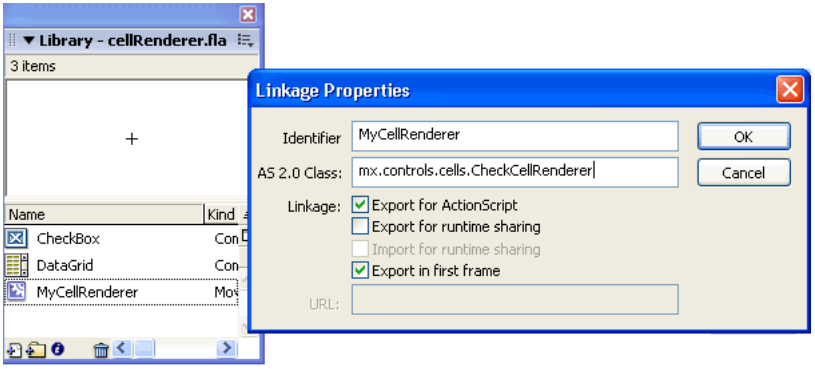
About the scrolling behavior of the List component

The List class uses a fairly complex algorithm for scrolling. A list only lays out as many rows as it can display at once; items beyond the value of the `rowCount` property don't get rows at all. When the list scrolls, it moves all the rows up or down (depending on the scrolling direction). The list then recycles the rows that are scrolled out of view; it reinitializes them and uses them for the new rows being scrolled into view. To do this, it sets the value of the old row to the new item in the view and moves the old row to where the new item is scrolled into view.

Because of this scrolling behavior, you cannot expect a cell to be used for only one value. Recycling of rows means that the cell renderer must know how to completely reset its state when it is set to a new value. For example, if your cell renderer creates an icon to display one item, it might need to remove that icon when another item is rendered with it. Assume your cell renderer is a container that will be filled with numerous item values over time, and it has to know how to completely change itself from displaying one value to displaying another. In fact, your cell should even know how to properly render undefined items, which might mean removing all old content in the cell.

Using the CellRenderer API

You must write a class with four methods (`CellRenderer.getPreferredHeight()`, `CellRenderer.getPreferredWidth()`, `CellRenderer.setSize()`, and `CellRenderer.setValue()`) that the list-based component uses to communicate with the cell. The class must be specified in the AS 2.0 Class text box in the Linkage Properties dialog box of a movie clip symbol in your Flash application. You can look at the CheckCellRenderer class that implements the Cell Renderer API for an example; it's located in FirstRun/classes/mx/controls/cells.



There are two methods and a property (`CellRenderer.getCellIndex()`, `CellRenderer.getDataLabel()`, and `CellRenderer.listOwner`) that are given automatically to a cell to allow it to communicate with the list-based component. For example, suppose a cell contains a check box that, when selected, causes a row to be selected. The cell renderer needs a reference to the list-based component that contains it in order to call the component's `selectedIndex` property. Also, the cell needs to know which item index it is currently rendering so that it can set `selectedIndex` to the correct number; the cell can use `CellRenderer.listOwner` and `CellRenderer.getCellIndex()` to do so. You do not need to implement these ActionScript elements; the cell receives them automatically when it is placed in the list-based component.

Simple cell renderer example

This section presents an example of a cell renderer that displays multiple lines of text in a cell.

A cell renderer class must implement the following methods:

- `CellRenderer.getPreferredHeight()`
- `CellRenderer.getPreferredWidth()`

  This method is necessary only for Menu components; otherwise, comment it out of the code, as in the example.
- `CellRenderer.setSize()`

  If a cell renderer class extends UIObject, implement `size()` instead.
- `CellRenderer.setValue()`

A cell renderer class must also declare the methods and property received from the List class:

- `CellRenderer.getCellIndex()`
- `CellRenderer.getDataLabel()`
- `CellRenderer.listOwner`

To test this cell renderer, create a Flash document with a list-based component. You must also create an empty movie clip and link it to the ActionScript 2.0 class MultiLineCell. Then, set the `cellRenderer` property of the component to the linkage identifier of the movie clip. For example, if you use a DataGrid component with the instance name `grid` and a movie clip with the linkage identifier `MultiLineCell`, the following code would cause the first column of the grid to render with multiline text:

```
grid.getColumnAt(0).cellRenderer = "MultiLineCell";
```

Note: Remember to add data to the DataGrid component.

If you were using a ComboBox component, you could write the following code:

```
comboBox.dropdown.cellRenderer = "MultiLineCell"
```

The following is the code for the MultiLineCell.as file:

```
class MultiLineCell extends mx.core.UIComponent
{
  var multiLineLabel; //the label to be used for text
  var owner; // the row that contains this cell
  var listOwner; // the List/grid/tree that contains this cell

  // empty constructor
  function MultiLineCell()
  {
  }

// UIObject expects you to fill in createChildren by instantiating
// all the movie clip assets you might need upon initialization.
// Here, it's just a label.
  function createChildren():Void
  {
// createLabel is a useful method of UIObject--all components use this
```

```
    var c = multiLineLabel = createLabel("multiLineLabel", 10);
// links the style of the label to the
// style of the grid
    c.styleName = listOwner;
    c.selectable = false;
    c.tabEnabled = false;
    c.background = false;
    c.border = false;
    c.multiline = true;
    c.wordWrap = true;
  }

// By extending UIComponent, you get setSize for free;
// however, UIComponent expects you to implement size().
// Assume __width and __height are set for you now.
// You're going to expand the cell to fit the whole rowHeight.
  function size():Void
  {
// __width and __height are the underlying variables
// of the getter/setters .width and .height
    var c = multiLineLabel;
    c._width = __width;
    c._height = __height;
  }

  function getPreferredHeight():Number
  {
  /* The cell is given a property, "owner",
  that references the row. It's always preferred
  that the cell take up most of the row's height.
  */
    return owner.__height - 4;
  }

  function setValue(suggested:String, item:Object, selected:Boolean):Void
  {
  // Set the text property of the label
  // You could also set the text property to a variable.
    multiLineLabel.text = "This text wraps to two lines!";
  }
  // function getPreferredWidth :: only necessary for menu
  // function getCellIndex :: not used in this cell renderer
  // function getDataLabel :: not used in this cell renderer
}
```

Methods to implement for the CellRenderer API

You must write a class with the following methods so that the List, DataGrid, Tree, or Menu component can communicate with the cell.

Method	Description
`CellRenderer.getPreferredHeight()`	Returns the preferred height of a cell.
`CellRenderer.getPreferredWidth()`	The preferred width of a cell.

Method	Description
CellRenderer.setSize()	Sets the width and height of a cell.
CellRenderer.setValue()	Sets the content to be displayed in the cell.

Methods provided by the CellRenderer API

The List, DataGrid, Tree, and Menu components give the following methods to the cell when it is created within the component. You do not need to implement these methods.

Method	Description
CellRenderer.getCellIndex()	Returns an object with two fields, columnIndex and rowIndex, that indicate the position of the cell.
CellRenderer.getDataLabel()	Returns a string containing the name of the cell renderer's data field.

Properties provided by the CellRenderer API

The List, DataGrid, Tree, and Menu component give the following properties to the cell when it is created within the component. You do not need to implement these properties.

Property	Description
CellRenderer.listOwner	A reference to the List component that contains the cell.
CellRenderer.owner	A reference to the row that contains the cell.

CellRenderer.getCellIndex()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.getCellIndex()

Parameters

None.

Returns

An object with two fields: columnIndex and itemIndex.

Description

Method; returns an object with two fields, columnIndex and itemIndex, that locate the cell in the component. Each field is an integer that indicates a cell's column position and item position. For any components other than the DataGrid component, the value of columnIndex is always 0.

This method is provided by the List class; you do not have to implement it. Declare it in your cell renderer class as follows, and use it in the functions in your cell renderer:

```
var getCellIndex:Function;
```

Example

This example edits a DataGrid component's data provider from within a cell:

```
var index = getCellIndex();
var colName = listOwner.getColumnAt(index.columnIndex).columnName;
listOwner.dataProvider.editField(index.itemIndex, colName, someVal);
```

CellRenderer.getDataLabel()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.getDataLabel()

Parameters

None.

Returns

A string.

Description

Method; returns a string containing the name of the cell renderer's data field. For the DataGrid component, this method returns the column name for the current cell.

This method is provided by the List class; you do not have to implement it. Declare it in your cell renderer class as follows, and use it in the functions in your cell renderer:

```
var getDataLabel:Function;
```

Example

The following code tells the cell that it's rendering the data field `"Price"`. The variable `p` is now equal to `"Price"`:

```
var p = getDataLabel();
```

CellRenderer.getPreferredHeight()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.getPreferredHeight()

Parameters

None.

Returns

The correct height for the cell.

Description

Method; returns the preferred height of a cell. This is especially important for getting the right height of text within the cell. If you set this value higher than the `rowHeight` property of the component, cells will bleed above and below the rows.

This method is not provided by the List class; you must implement it. It tells the rows of the list how to center the cell and how to adjust the cell's height if necessary. If necessary, you can return a constant (for example, 22), or you can measure and return the height of the contents. You can also return `owner.height`, which is the height of the row.

Example

This example returns the value 20, which indicates that the cell should be 20 pixels high:

```
function getPreferredHeight(Void) :Number
{
   return 20;
}
```

This example returns a value that is 4 pixels less that the height of the row:

```
function getPreferredHeight():Number
{
/* You know the cell is given a property, "owner", which is the row. It's
  always preferred for the cell to take up most of the row's height.
*/
return owner.__height - 4;
}
```

CellRenderer.getPreferredWidth()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.getPreferredWidth()

Parameters

None.

Returns

A value (of type Number) that indicates the correct width of the cell.

Description

Method; the preferred width of a cell. If you specify a width greater than that of the component, the cell may be cut off.

You need to implement this method only for the Menu component. Your cell will be sized to whatever the width of the row is, except in a menu, which must measure the text for the width of the row.

Example

This example returns the value 3, which indicates that the cell should be three times bigger than the length of the string it is rendering:

```
function getPreferredWidth():Number
{
   return myString.length*3;
}
```

This example comments out the `getPreferredWidth()` method:

```
// function getPreferredWidth :: only really necessary for a menu
```

CellRenderer.listOwner

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.listOwner

Description

Property; a reference to the list that owns the cell. That list can be a DataGrid, Tree, List, or Menu component.

This method is provided by the List class; you do not have to implement it. Declare it in your cell renderer class as follows, and use it as a reference back to the list (or tree, menu, or grid):

```
var listOwner:MovieClip; // or UIObject, etc.
```

Example

This example finds the list's selected item in a cell:

```
var s = listOwner.selectedItem;
```

CellRenderer.owner

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.owner
```

Description

Property; a reference to the row that contains the cell.

This method is provided by the List class; you do not have to implement it. Declare it in your cell renderer class and use it as a reference:

```
var owner:MovieClip; // or UIObject, etc.
```

CellRenderer.setSize()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.setSize(width, height)
```

Parameters

width A number that indicates the width at which to lay out the component.

height A number that indicates the height at which to lay out the component.

Returns

Nothing.

Description

Method; lets the list tell its cells the size at which they should lay themselves out. The cell renderer should do layout so that it fits in the specified area, or the cell may bleed into other parts of the list and appear broken.

If the cell renderer extends the UIObject class, you should implement the `size()` method instead. Write the same function that you would write for `setSize()`, but use the `width` and `height` properties instead of parameters.

Example

This example sizes an image in the cell to fit within the bounds specified by the list:

```
function setSize(w:Number, h:Number):Void
{
   image._width = w-2;
   image._height = h-2;
   image._x = image._y = 1;
}
```

This example is in a cell renderer class that extends UIComponent (which extends UIObject), so you must implement `size()` instead of `setSize()`, as follows:

```
// By extending UIComponent, you get setSize for free;
// however, UIComponent expects you to implement size().
// Assume __width and __height are set for you now.
// You're going to expand the cell to fit the whole rowHeight.

function size():Void
{
// __width and __height are the underlying variables
// of the getters/setters .width and .height.
  var c = multiLineLabel;
  c._width = __width;
  c._height = __height;
}
```

CellRenderer.setValue()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.setValue(*suggested*, *item*, *selected*)

Parameters

suggested A value to be used for the cell renderer's text, if any is needed.

item An object that is the entire item to be rendered. The cell renderer can use properties of this object for rendering.

selected A string with the following possible values: `"normal"`, `"highlighted"`, and `"selected"`.

Returns

Nothing.

Description

Method; takes the values given and creates a representation of them in the cell. This resolves any difference between what was displayed in the cell and what needs to be displayed in the cell for the new item. (Remember that any cell could display many values during its time in the list.) This is the most important CellRenderer method, and you must implement it in every cell renderer.

The `setValue()` method is called frequently (for example, when a rollover, a selection, column resizing, or scrolling occurs). Therefore, you should write `if` statements in the body of `setValue()` that allow code to run only if a change has occurred. See the "Example" section below.

If a row is selected and the mouse pointer is over it, the value of the selected parameter is `"highlighted"`, not `"selected"`. This can cause problems if you're trying to make the cell renderer behave differently according to whether the row is in a selected state. To test whether the current row is in a selected state, use the following code:

```
var reallySelected:Boolean = selected ne "normal" && listOwner.selectedNode ==
  item;
```

Example

The following example shows how to use `setValue()` and `editField()` to reference a cell renderer instance in a grid.

Because a particular cell might not exist on the Stage (for example, if it's scrolled out of the display area) or because it might be reused to render another value, you cannot directly reference a specific cell renderer instance in the grid.

Instead, use the data provider to communicate with a specific cell in the grid. The data provider holds all the state information about the grid. To display a given cell as enabled or selected (checked), there should be a corresponding field in the data provider to hold that information. The `setValue()` method of your cell renderer communicates changes in the data provider's state to the cell. The following is a `setValue()` implementation from a theoretical cell renderer that renders a check box in the cells:

```
function setValue(str, itm, sel)
{
/* Assume the data provider has two relevant fields for this cell : checked and
  enabled.
The form of such a data provider might look like this:
[
{field1:"DisplayMe", field2:"SomeString", checked:true, enabled:false}
{field1:"DisplayMe", field2:"SomeString", checked:false, enabled:true}
{field1:"DisplayMe", field2:"SomeString", checked:true, enabled:true}
]
*/

// redundancy checking
  if (myCheck.selected!=itm.checked){
      myCheck.selected = itm.checked;
  }
```

```
  if (myCheck.enabled!=itm.enabled){
      myCheck.enabled = itm.enabled;
  }
}
```

If you want to enable the check box on the second row, you communicate through the data provider. Any change to the data provider (when made through a DataProvider method such as `DataProvider.editField()`) calls `setValue()` to refresh the display of the grid. This code would be written in the Flash application, either on a frame, on an object, or in another class file (but not in the cell renderer class file):

```
// calls setValue() again
myGrid.editField(1, "enabled", true);
```

The following example loads an image in a loader component within the cell, depending on the value passed:

```
function setValue(suggested, item, selected) : Void
{
   // clear the loader
   loader.contentPath = undefined;
   // the list has URLs for different images in its data provider
   if (suggested!=undefined)
   loader.contentPath = suggested;
}
```

The following example is from a multiline text cell renderer:

```
function setValue(suggested:String, item:Object, selected:Boolean):Void
{
  // adds the text to the label
  multiLineLabel.text = suggested;
}
```

CheckBox component

A check box is a square box that can be selected or deselected. When it is selected, a check mark appears in the box. You can add a text label to a check box and place it to the left, right, top, or bottom.

A check box can be enabled or disabled in an application. If a check box is enabled and a user clicks it or its label, the check box receives input focus and displays its pressed appearance. If a user moves the pointer outside the bounding area of a check box or its label while pressing the mouse button, the component's appearance returns to its original state and it retains input focus. The state of a check box does not change until the mouse is released over the component. Additionally, the check box has two disabled states, selected and deselected, which do not allow mouse or keyboard interaction.

If a check box is disabled, it displays its disabled appearance, regardless of user interaction. In the disabled state, a button doesn't receive mouse or keyboard input.

A CheckBox instance receives focus if a user clicks it or tabs to it. When a CheckBox instance has focus, you can use the following keys to control it:

Key	Description
Shift+Tab	Moves focus to the previous element.
Spacebar	Selects or deselects the component and triggers the `click` event.
Tab	Moves focus to the next element.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each CheckBox instance reflects changes made to parameters in the Property inspector or Component inspector during authoring.

When you add the CheckBox component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code to enable accessibility:

```
mx.accessibility.CheckBoxAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the CheckBox component

A check box is a fundamental part of any form or web application. You can use check boxes wherever you need to gather a set of `true` or `false` values that aren't mutually exclusive. For example, a form collecting personal information about a customer could have a list of hobbies for the customer to select; each hobby would have a check box beside it.

CheckBox parameters

You can set the following authoring parameters for each CheckBox component instance in the Property inspector or in the Component inspector:

label sets the value of the text on the check box; the default value is `defaultValue`.

selected sets the initial value of the check box to checked (`true`) or unchecked (`false`).

labelPlacement orients the label text on the check box. This parameter can be one of four values: `left`, `right`, `top`, or `bottom`; the default value is `right`. For more information, see `CheckBox.labelPlacement`.

You can write ActionScript to control these and additional options for the CheckBox component using its properties, methods, and events. For more information, see "CheckBox class" on page 161.

Creating an application with the CheckBox component

The following procedure explains how to add a CheckBox component to an application while authoring. The following example is a form for an online dating application. The form is a query that searches for possible dating matches for the customer. The query form must have a check box labeled Restrict Age that permits customers to restrict their search to a specified age group. When the Restrict Age check box is selected, the customer can then enter the minimum and maximum ages into two text fields. (These text fields are enabled only when the check box is selected.)

To create an application with the CheckBox component:

1. Drag two TextInput components from the Components panel to the Stage.
2. In the Property inspector, enter the instance names **minimumAge** and **maximumAge**.
3. Drag a CheckBox component from the Components panel to the Stage.
4. In the Property inspector, do the following:
   - Enter **restrictAge** for the instance name.
   - Enter **Restrict Age** for the label parameter.
5. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
restrictAgeListener = new Object();
restrictAgeListener.click = function (evt){
  minimumAge.enabled = evt.target.selected;
  maximumAge.enabled = evt.target.selected;
}
restrictAge.addEventListener("click", restrictAgeListener);
```

This code creates a `click` event handler that enables and disables the `minimumAge` and `maximumAge` text field components, which have already been placed on Stage. For more information, see `CheckBox.click` and "TextInput component" on page 742.

Customizing the CheckBox component

You can transform a CheckBox component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (`UIObject.setSize()`) or any applicable properties and methods of the CheckBox class. Resizing the check box does not change the size of the label or the check box icon; it only changes the size of the bounding box.

The bounding box of a CheckBox instance is invisible and also designates the hit area for the instance. If you increase the size of the instance, you also increase the size of the hit area. If the bounding box is too small to fit the label, the label is clipped to fit.

Using styles with the CheckBox component

You can set style properties to change the appearance of a CheckBox instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

A CheckBox component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.

Style	Theme	Description
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".
symbolBackgroundColor	Sample	The background color of the check box. The default value is 0xFFFFFF (white).
symbolBackgroundDisabledColor	Sample	The background color of the check box when disabled. The default value is 0xEFEEEF (light gray).
symbolBackgroundPressedColor	Sample	The background color of the check box when pressed. The default value is 0xFFFFFF (white).
symbolColor	Sample	The color of the check mark. The default value is 0x000000 (black).
symbolDisabledColor	Sample	The color of the disabled check mark. The default value is 0x848384 (dark gray).

Using skins with the CheckBox component

The CheckBox component uses symbols in the library to represent the button states. To skin the CheckBox component while authoring, modify symbols in the Library panel. The CheckBox component skins are located in the Flash UI Components 2/Themes/MMDefault/CheckBox Assets/states folder in the library of either the HaloTheme.fla file or the SampleTheme.fla file. For more information, see "About skinning components" on page 80.

A CheckBox component uses the following skin properties:

Property	Description
falseUpSkin	The up (normal) unchecked state. The default is CheckFalseUp.
falseDownSkin	The pressed unchecked state. The default is CheckFalseDown.
falseOverSkin	The over unchecked state. The default is CheckFalseOver.
falseDisabledSkin	The disabled unchecked state. The default is CheckFalseDisabled.
trueUpSkin	The toggled checked state. The default is CheckTrueUp.
trueDownSkin	The pressed checked state. The default is CheckTrueDown.
trueOverSkin	The over checked state. The default is CheckTrueOver.
trueDisabledSkin	The disabled checked state. The default is CheckTrueDisabled.

Each of these skins corresponds to the icon indicating the CheckBox state. The CheckBox component does not have a border or background.

To create movie clip symbols for CheckBox skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.

3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the CheckBox Assets folder to the library for your document.
4. Expand the CheckBox Assets/States folder in the library of your document.
5. Open the symbols you want to customize for editing.

   For example, open the CheckFalseDisabled symbol.
6. Customize the symbol as desired.

   For example, change the inner white square to a light gray.
7. Repeat steps 5-6 for all symbols you want to customize.

   For example, repeat the color change for the inner box of the CheckTrueDisabled symbol.
8. Click the Back button to return to the main Timeline.
9. Drag a CheckBox component to the Stage.

   For this example, drag two instances to show the two new skin symbols.
10. Set the CheckBox instance properties as desired.

    For this example, set one CheckBox instance to `true`, and use ActionScript to set both CheckBox instances to disabled.
11. Select Control > Test Movie.

CheckBox class

Inheritance MovieClip > UIObject class > UIComponent class > SimpleButton class > Button component > CheckBox

ActionScript Class Name mx.controls.CheckBox

The properties of the CheckBox class let you create a text label and position it to the left, right, top, or bottom of a check box at runtime.

Setting a property of the CheckBox class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

The CheckBox component uses the Focus Manager to override the default Flash Player focus rectangle and draw a custom focus rectangle with rounded corners. For more information, see "Creating custom focus navigation" on page 50.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.CheckBox.version);
```

Note: `The code trace(myCheckBoxInstance.version);` returns `undefined`.

Method summary for the CheckBox class

There are no methods exclusive to the CheckBox class.

Methods inherited from the UIObject class

The following table lists the methods the CheckBox class inherits from the UIObject class. When calling these methods from the CheckBox object, use the form *checkBoxInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the CheckBox class inherits from the UIComponent class. When calling these methods from the CheckBox object, use the form *checkBoxInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the CheckBox class

The following table lists properties of the CheckBox class.

Property	Description
CheckBox.label	Specifies the text that appears next to a check box.
CheckBox.labelPlacement	Specifies the orientation of the label text in relation to a check box.
CheckBox.selected	Specifies whether the check box is selected (`true`) or deselected (`false`).

Properties inherited from the UIObject class

The following table lists the properties the CheckBox class inherits from the UIObject class. When accessing these properties from the CheckBox object, use the form *checkBoxInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the CheckBox class inherits from the UIComponent class. When accessing these properties from the CheckBox object, use the form *checkBoxInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the SimpleButton class

The following table lists the properties the CheckBox class inherits from the SimpleButton class. When accessing these properties from the CheckBox object, use the form *checkBoxInstance.propertyName*.

Property	Description
SimpleButton.emphasized	Indicates whether a button has the appearance of a default push button.
SimpleButton.emphasizedStyleDeclaration	The style declaration when the emphasized property is set to true.
SimpleButton.selected	A Boolean value indicating whether the button is selected (true) or not (false). The default value is false.
SimpleButton.toggle	A Boolean value indicating whether the button behaves as a toggle switch (true) or not (false). The default value is false.

Properties inherited from the Button class

The following table lists the properties the CheckBox class inherits from the Button class. When accessing these properties from the CheckBox object, use the form *checkBoxInstance.propertyName*.

Property	Description
Button.label	Specifies the text that appears in a button.
Button.labelPlacement	Specifies the orientation of the label text in relation to an icon.

Event summary for the CheckBox class

The following table lists an event of the CheckBox class.

Event	Description
CheckBox.click	Triggered when the mouse is clicked (released) over the check box, or if the check box has focus and the Spacebar is pressed.

Events inherited from the UIObject class

The following table lists the events the CheckBox class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.

Event	Description
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the CheckBox class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Events inherited from the SimpleButton class

The following table lists the event the CheckBox class inherits from the SimpleButton class.

Event	Description
SimpleButton.click	Broadcast when a button is clicked.

CheckBox.click

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(click){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.click = function(eventObject){
   ...
}
checkBoxInstance.addEventListener("click", listenerObject)
```

Description

Event; broadcast to all registered listeners when the mouse is clicked (released) over the check box, or if the check box has focus and the Spacebar is pressed.

The first usage example uses an `on()` handler and must be attached directly to a CheckBox instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the check box `myCheckBox`, sends "_level0.myCheckBox" to the Output panel:

```
on(click){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`checkBoxInstance`*) dispatches an event (in this case, `click`), and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. The event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `addEventListener()` method (see `EventDispatcher.addEventListener()`) on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a button called `checkBoxInstance` is clicked. The first line of code creates a listener object called `form`. The second line defines a function for the `click` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function (in this example, `eventObj`) to generate a message. The `target` property of an event object is the component that generated the event (in this example, `checkBoxInstance`). The `CheckBox.selected` property is accessed from the event object's `target` property. The last line calls `addEventListener()` from `checkBoxInstance` and passes it the `click` event and the `form` listener object as parameters.

```
form = new Object();
form.click = function(eventObj){
  trace("The selected property has changed to " + eventObj.target.selected);
}
checkBoxInstance.addEventListener("click", form);
```

The following code also sends a message to the Output panel when `checkBoxInstance` is clicked. The `on()` handler must be attached directly to `checkBoxInstance`:

```
on(click){
  trace("check box component was clicked");
}
```

See also

`EventDispatcher.addEventListener()`

CheckBox.label

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

checkBoxInstance.label

Description

Property; indicates the text label for the check box. By default, the label appears to the right of the check box. Setting this property overrides the label parameter specified in the Parameters tab of the Component Inspector panel.

Example

The following code sets the text that appears beside the CheckBox component and sends the value to the Output panel:

```
checkBox.label = "Remove from list";
trace(checkBox.label)
```

See also

CheckBox.labelPlacement

CheckBox.labelPlacement

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

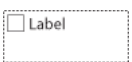
Usage

checkBoxInstance.labelPlacement

Description

Property; a string that indicates the position of the label in relation to the check box. The following are the four possible values (the dotted lines represent the bounding area of the component; they are invisible in a document):

- `"right"` The check box is pinned to the upper left corner of the bounding area. The label is set to the right of the check box. This is the default value.



- `"left"` The check box is pinned to the upper right corner of the bounding area. The label is set to the left of the check box.

- `"bottom"` The label is set below the check box. The check box and label are centered horizontally and vertically.

- `"top"` The label is placed below the check box. The check box and label are centered horizontally and vertically.

You can change the bounding area of a component while authoring by using the Transform command or at runtime using the `UIObject.setSize()` property. For more information, see "Customizing the CheckBox component" on page 159.

Example

The following example sets the placement of the label to the left of the check box:

```
checkBox_mc.labelPlacement = "left";
```

See also

`CheckBox.label`

CheckBox.selected

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`checkBoxInstance``.selected`

Description

Property; a Boolean value that selects (`true`) or deselects (`false`) the check box.

Example

The following example selects the instance `checkbox1`:

```
checkbox1.selected = true;
```

Collection interface (Flash Professional only)

ActionScript Class Name mx.utils.Collection

The collection class is distributed in the common classes library as a compiled clip symbol. To access this class, select Window > Other Panels > Common Libraries > Classes > UtilsClasses.

The collection interface lets you programmatically manage a group of related items, called *collection items*. Each collection item in this set has properties that are described in the metadata of the collection item class definition.

Components can expose properties as collections, which you can manipulate while authoring by using the Values dialog box from the Component inspector. Using this dialog box, you can add items, remove items, change properties of items, and change the position of items within the collection. For more information on collections and collection items, see "About the Collection tag" on page 942.

You typically use the collection interface with components that use the Collection metadata tag to create collection properties. Although you can create, access, and delete Collection instances programmatically, collections are most often used in the context of a component. Flash MX Professional 2004 provides implementations of both collection-related interfaces (CollectionImpl for Collection, and IteratorImpl for Iterator).

Method summary for the Collection interface

The following table lists the methods of the Collection interface.

Method	Description
`Collection.addItem()`	Adds a new item to the end of the collection.
`Collection.contains()`	Indicates whether the collection contains the specified item.
`Collection.clear()`	Removes all elements from the collection.
`Collection.getItemAt()`	Returns an item within the collection by using its index.
`Collection.getIterator()`	Returns an iterator over the elements in the collection.
`Collection.getLength()`	Returns the number of items in the collection.
`Collection.isEmpty()`	Indicates whether the collection is empty.
`Collection.removeItem()`	Removes the specified item from the collection.

Collection.addItem()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
collection.addItem(item)
```

Parameters

item The object to be added to the collection. If *item* is `null`, it is not added to the collection.

Returns

A Boolean value of `true` if the collection was changed as a result of the operation.

Description

Method; adds a new item to the end of the collection.

Example

The following example calls `addItem()`:

```
on (click) {
import CompactDisc;

  var myColl:mx.utils.Collection;
  myColl = _parent.thisShelf.MyCompactDiscs;
  myCD = new CompactDisc();
  myCD.Artist = "John Coltrane";
  myCD.Title = "Giant Steps";

  var wasAdded:Boolean = myColl.addItem(myCD);
}
```

Collection.contains()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

collection.contains(*item*)

Parameters

item The object whose presence in the collection is to be tested.

Returns

A Boolean value of `true` if the collection contains *item*.

Description

Method; indicates whether the collection contains the specified item. For Flash to consider the objects as equal, they must refer to the same object. If *item* is a different object, `Collection.contains()` returns `false`, even if the object's properties are all equal.

Example

The following example calls `contains()`:

```
var myColl:mx.utils.Collection;
myColl = _parent.thisShelf.MyCompactDiscs;

var itr:mx.utils.Iterator = myColl.getIterator();
while (itr.hasNext()) {
  var cd:CompactDisc = CompactDisc(itr.next());
  var title:String = cd.Title;
  var artist:String = cd.Artist;

  if(myColl.contains(cd)) {
    trace("myColl contains " + title);
  }
  else {
    trace("myColl does not contain " + title);
  }
}
```

Collection.clear()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

collection.clear()

Returns

Nothing.

Description

Method; removes all of the elements from the collection.

Example

The following example calls `clear()`:

```
on (click) {
  var myColl:mx.utils.Collection;
  myColl = _parent.thisShelf.MyCompactDiscs;
  myColl.clear();
}
```

Collection.getItemAt()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

collection.getItemAt(*index*)

Parameters

index A number that indicates the location of *item* within the collection. This is a zero-based index, so 0 retrieves the first item, 1 retrieves the second item, and so on.

Returns

An object containing a reference to the specified collection item, or `null` if *index* is out of bounds.

Description

Method; returns an item within the collection by using its index.

Example

The following example calls `getItemAt()`:

```
//...
  var myColl:mx.utils.Collection;
  myColl = _parent.thisShelf.MyCompactDiscs;
  var myCD = CompactDisc(myColl.getItemAt(0));
  if (myCD !=null) {
    trace("Retrieved " + myCD.Title);
  }
//...
```

Collection.getIterator()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

collection.getIterator()

Returns

An Iterator object that you can use to step through the collection.

Description

Method; returns an iterator over the elements in the collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of a class that provides a guarantee).

Example

The following example calls `getIterator()`:

```
on (click) {
  var myColl:mx.utils.Collection;
  myColl = _parent.thisShelf.MyCompactDiscs;

  var itr:mx.utils.Iterator = myColl.getIterator();
  while (itr.hasNext()) {
    var cd:CompactDisk = CompactDisc(itr.next());
    var title:String = cd.Title;
    var artist:String = cd.Artist;

    trace("Title: " + title + " - Artist: " + artist);
  }
}
```

Collection.getLength()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

collection.getLength()

Returns

The number of items in the collection.

Description

Method; returns the number of items in the collection.

Example

The following example calls `getLength()`:

```
//...
var myColl:mx.utils.Collection;
myColl = _parent.thisShelf.MyCompactDiscs;
trace ("Collection size is: " + myColl.getLength());
//...
```

Collection.isEmpty()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

collection.isEmpty()

Returns

A Boolean value of `true` if the collection is empty.

Description

Method; indicates whether the collection is empty.

Example

The following example calls `isEmpty()`:

```
on (click) {
  var myColl:mx.utils.Collection;
  myColl = _parent.thisShelf.MyCompactDiscs;
  if (myColl.isEmpty()) {
    trace("No CDs in the collection");
  }
}
//...
```

Collection.removeItem()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

collection.removeItem(*item*)

Parameters

item The object to be removed from the collection.

Returns

A Boolean value of `true` if *item* was removed successfully.

Description

Method; removes the specified item from the collection. Because `Collection.removeItem()` dynamically reduces the size of the collection, do not call this method while looping through an iterator.

Example

The following example calls `removeItem()`:

```
var myColl:mx.utils.Collection;
myColl = _parent.thisShelf.MyCompactDiscs;

// get this from a text input box
var removeArtist:String = _parent.tArtistToRemove.text;
var removeSize:Number = 0;

if (myColl.isEmpty()) {
  trace("No CDs in the collection");
}
else {
  var toRemove:Array = new Array();
  var itr:mx.utils.Iterator = myColl.getIterator();
  var cd:CompactDisc = new CompactDisc();
  var title:String = "";
  var artist:String = "";
  while (itr.hasNext()) {
    cd = CompactDisc(itr.next());
    title = cd.Title;
    artist = cd.Artist;
    if(artist == removeArtist) {
      // mark this artist for deletion
      removeSize = toRemove.push(cd);
      trace("*** Marked for deletion: " + artist + "|" + title);
    }
  }
  // after while loop, remove the bad ones
  var removeCD:CompactDisc = new CompactDisc();
  for(i = 0; i < removeSize; i++) {
    removeCD = toRemove[i];
    trace("Removing: " + removeCD.Artist + "|" + removeCD.Title);
    myColl.removeItem(removeCD);
  }
}
```

ComboBox component

A combo box allows a user to make a single selection from a drop-down list. A combo box can be static or editable. An editable combo box allows a user to enter text directly into a text field at the top of the list, as well as selecting an item from a drop-down list. If the drop-down list hits the bottom of the document, it opens up instead of down. The combo box is composed of three subcomponents: a Button component, a TextInput component, and a List component.

When a selection is made in the list, the label of the selection is copied to the text field at the top of the combo box. It doesn't matter if the selection is made with the mouse or the keyboard.

A ComboBox component receives focus if you click the text box or the button. When a ComboBox component has focus and is editable, all keystrokes go to the text box and are handled according to the rules of the TextInput component (see "TextInput component" on page 742), with the exception of the following keys:

Key	Description
Control+Down Arrow	Opens the drop-down list and gives it focus.
Shift+Tab	Moves focus to the previous object.
Tab	Moves focus to the next object.

When a ComboBox component has focus and is static, 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 static combo box:

Key	Description
Control+Down Arrow	Opens the drop-down list and gives it focus.
Control+Up Arrow	Closes the drop-down list, if open in the stand-alone and browser versions of Flash Player.
Down Arrow	Moves the selection down one item.
End	Selection moves to the bottom of the list.
Escape	Closes the drop-down list and returns focus to the combo box in Test Movie mode.
Enter	Closes the drop-down list and returns focus to the combo box.
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.
Shift+Tab	Moves focus to the previous object.
Tab	Moves focus to the next object.

When the drop-down list of a combo box 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 Arrow	If the drop-down list is open, focus returns to the text box and the drop-down list closes in the stand-alone and browser versions of Flash Player.
Down Arrow	Moves the selection down one item.
End	Moves the insertion point to the end of the text box.
Enter	If the drop-down list is open, focus returns to the text box and the drop-down list closes.
Escape	If the drop-down list is open, focus returns to the text box and the drop-down list closes in Test Movie mode.
Home	Moves the insertion point to the beginning of the text box.
Page Down	Moves the selection down one page.
Page Up	Moves the selection up one page.
Tab	Moves focus to the next object.
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 focus to the previous object.
Up Arrow	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 ten-line drop-down list will show items 0-9, 9-18, 18-27, and so on, with one item overlapping per page.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each ComboBox component instance on the Stage reflects changes made to parameters in the Property inspector or Component inspector during authoring. However, the drop-down list does not open in the live preview, and the first item displays as the selected item.

When you add the ComboBox component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code to enable accessibility:

```
mx.accessibility.ComboBoxAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the ComboBox component

You can use a ComboBox component in any form or application that requires a single choice from a list. For example, you could provide a drop-down list of states in a customer address form. You can use an editable combo box for more complex scenarios. For example, in an application that provides driving directions, you could use an editable combo box for a user to enter her origin and destination addresses. The drop-down list would contain her previously entered addresses.

ComboBox parameters

You can set the following authoring parameters for each ComboBox component instance in the Property inspector or in the Component inspector:

editable determines if the ComboBox component is editable (`true`) or only selectable (`false`). The default value is `false`.

labels populates the ComboBox component with an array of text values.

data associates a data value with each item in the ComboBox component. The data parameter is an array.

rowCount sets the maximum number of items that can be displayed in the list. The default value is 5.

You can write ActionScript to set additional options for ComboBox instances using the methods, properties, and events of the ComboBox class. For more information, see "ComboBox class" on page 182.

Creating an application with the ComboBox component

The following procedure explains how to add a ComboBox component to an application while authoring. In this example, the combo box presents a list of cities to select from in its drop-down list.

To create an application with the ComboBox component:

1. Drag a ComboBox component from the Components panel to the Stage.
2. Select the Transform tool and resize the component on the Stage.

   The combo box can only be resized on the Stage during authoring. Typically, you would only change the width of a combo box to fit its entries.
3. Select the combo box and, in the Property inspector, enter the instance name **comboBox**.
4. In the Component inspector or Property inspector, do the following:
   - Enter **Minneapolis**, **Portland**, and **Keene** for the label parameter. Double-click the label parameter field to open the Values dialog box. Then click the plus sign to add items.
   - Enter **MN.swf**, **OR.swf**, and **NH.swf** for the data parameter.

     These are imaginary SWF files that, for example, you could load when a user selects a city from the combo box.

5. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
function change(evt){
  trace(evt.target.selectedItem.label);
}
comboBox.addEventListener("change", this);
```

The last line of code adds a `change` event handler to the `ComboBox` instance. For more information, see `ComboBox.change`.

Customizing the ComboBox component

You can transform a ComboBox component horizontally and vertically while authoring. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands.

If text is too long to fit in the combo box, the text is clipped to fit. You must resize the combo box while authoring to fit the label text.

In editable combo boxes, only the button is the hit area—not the text box. For static combo boxes, the button and the text box constitute the hit area. The hit area responds by opening or closing the drop-down list.

Using styles with the ComboBox component

You can set style properties to change the appearance of a ComboBox component. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

The combo box has two unique styles: `openDuration` and `openEasing`. Other styles are passed to the button, text box, and drop-down list of the combo box through those individual components, as follows:

- The button is a Button instance and uses its styles. (See "Using styles with the Button component" on page 133.)
- The text is a TextInput instance and uses its styles. (See "Using styles with the TextInput component" on page 744.)
- The drop-down list is an List instance and uses its styles. (See "Using styles with the List component" on page 453.)

A ComboBox component uses the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`backgroundColor`	Both	The background color. The default color is white.

Style	Theme	Description
border styles	Both	The Button subcomponent uses two RectBorder instances for its borders and responds to the styles defined on that class. See "RectBorder class" on page 647. In the Halo theme, the ComboBox component uses a custom rounded border for the collapsed portion of the ComboBox. The colors of this portion of the ComboBox can be modified only through skinning. See "Using skins with the ComboBox component" on page 181.
color	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
disabledColor	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
embedFonts	Both	Boolean value that indicates whether the font specified in fontFamily is an embedded font. This style must be set to true if fontFamily refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to true and fontFamily does not refer to an embedded font, no text will be displayed. The default value is false.
fontFamily	Both	The font name for text. The default value is "_sans".
fontSize	Both	The point size for the font. The default value is 10.
fontStyle	Both	The font style: either "normal" or "italic". The default value is "normal".
fontWeight	Both	The font weight: either "none" or "bold". The default value is "none". All components can also accept the value "normal" in place of "none" during a setStyle() call, but subsequent calls to getStyle() will return "none".
textAlign	Both	The text alignment: either "left", "right", or "center". The default value is "left".
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".
openDuration	Both	The duration, in milliseconds, of the transition animation. The default value is 250.
openEasing	Both	A reference to a tweening function that controls the animation. Defaults to sine in/out. For more information, see "Customizing component animations" on page 75.

The following example demonstrates how to use List styles to control the behavior of the drop-down portion of a ComboBox component.

```
// comboBox is an instance of the ComboBox component on Stage
comboBox.setStyle("alternatingRowColors", [0xFFFFFF, 0xBFBFBF]);
```

Using skins with the ComboBox component

The ComboBox component uses symbols in the library to represent the button states and has skin variables for the down arrow. These skins are located in the Flash UI Components 2/Themes/MMDefault/ComboBox Assets/States folder of the HaloTheme.fla and SampleTheme.fla files. The information below describes these skins and provides steps for customizing them.

The ComboBox component also uses scroll bar skins for the drop-down list's scroll bar and two RectBorder class instances for the border around the text input and drop-down list. For information on customizing these skins, see "Using skins with the UIScrollBar component" on page 831 and "RectBorder class" on page 647. For more information on the methods available to skin components, see "About skinning components" on page 80.

A ComboBox component uses the following skin properties:

Property	Description
`ComboDownArrowDisabledName`	The down arrow's disabled state. The default is `ComboDownArrowDisabled`.
`ComboDownArrowDownName`	The down arrow's down state. The default is `ComboDownArrowDown`.
`ComboDownArrowUpName`	The down arrow's up state. The default is `ComboDownArrowOver`.
`ComboDownArrowOverName`	The down arrow's over state. The default is `ComboDownArrowUp`.

To create movie clip symbols for ComboBox skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the ComboBox Assets folder to the library for your document.
4. Expand the ComboBox Assets/States folder in the library of your document.
5. Open the symbols you want to customize for editing.

   For example, open the ComboDownArrowDisabled symbol.
6. Customize the symbol as desired.

   For example, change the inner white square to a light gray.
7. Repeat steps 5-6 for all symbols you want to customize.
8. Click the Back button to return to the main Timeline.
9. Drag a ComboBox component to the Stage.
10. Set the ComboBox instance properties as desired.

    For this example, use ActionScript to set the ComboBox to disabled.
11. Select Control > Test Movie.

ComboBox class

Inheritance MovieClip > UIObject class > UIComponent class > ComboBase > ComboBox

ActionScript Class Name mx.controls.ComboBox

The ComboBox component combines three separate subcomponents: Button, TextInput, and List. Most of the methods, properties, and events of each subcomponent are available directly from the ComboBox component and are listed in the summary tables for the ComboBox class.

The drop-down list in a combo box is provided either as an array or as a data provider. If you use a data provider, the list changes at runtime. You can change the source of the ComboBox data dynamically by switching to a new array or data provider.

Items in a combo box list are indexed by position, starting with the number 0. An item can be one of the following:

- A primitive data type.
- An object that contains a `label` property and a `data` property

  Note: An object may use the `ComboBox.labelFunction` or `ComboBox.labelField` property to determine the `label` property.

If the item is a primitive data type other than String, it is converted to a string. If an item is an object, the `label` property must be a string and the `data` property can be any ActionScript value.

ComboBox methods to which you supply items have two parameters, *`label`* and *`data`*, that refer to the properties above. Methods that return an item return it as an object.

A combo box defers the instantiation of its drop-down list until a user interacts with it. Therefore, a combo box may appear to respond slowly on first use.

Use the following code to programmatically access the ComboBox component's drop-down list and override the delay:

```
var foo = myComboBox.dropdown;
```

Accessing the drop-down list may cause a pause in the application. This may occur when the user first interacts with the combo box, or when the above code runs.

Method summary for the ComboBox class

The following table lists methods of the ComboBox class.

Method	Description
`ComboBox.addItem()`	Adds an item to the end of the list.
`ComboBox.addItemAt()`	Adds an item to the end of the list at the specified index.
`ComboBox.close()`	Closes the drop-down list.
`ComboBox.getItemAt()`	Returns the item at the specified index.
`ComboBox.open()`	Opens the drop-down list.
`ComboBox.removeAll()`	Removes all items in the list.

Method	Description
ComboBox.removeItemAt()	Removes an item from the list at the specified location.
ComboBox.replaceItemAt()	Replaces the content of the item at the specified index.
ComboBox.sortItems()	Sorts the list using a compare function.
ComboBox.sortItemsBy()	Sorts the list using a field of each item.

Methods inherited from the UIObject class

The following table lists the methods the ComboBox class inherits from the UIObject class. When calling these methods from the ComboBox object, use the form *comboBoxInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the ComboBox class inherits from the UIComponent class. When calling these methods from the ComboBox object, use the form *comboBoxInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the ComboBox class

The following table lists properties of the ComboBox class.

Property	Description
ComboBox.dataProvider	The data model for the items in the list.
ComboBox.dropdown	Returns a reference to the List component contained by the combo box.
ComboBox.dropdownWidth	The width of the drop-down list, in pixels.
ComboBox.editable	Indicates whether a combo box is editable.
ComboBox.labelField	Indicates which data field to use as the label for the drop-down list.
ComboBox.labelFunction	Specifies a function to compute the label field for the drop-down list.
ComboBox.length	Read-only; the length of the drop-down list.
ComboBox.restrict	The set of characters that a user can enter in the text field of a combo box.
ComboBox.rowCount	The maximum number of list items to display at one time.
ComboBox.selectedIndex	The index of the selected item in the drop-down list.
ComboBox.selectedItem	The value of the selected item in the drop-down list.
ComboBox.text	The string of text in the text box.
ComboBox.textField	A reference to the TextInput component in the combo box.
ComboBox.value	The value of the text box (editable) or drop-down list (static).

Properties inherited from the UIObject class

The following table lists the properties the ComboBox class inherits from the UIObject class. When accessing these properties from the ComboBox object, use the form *comboBoxInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.

Property	Description
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the ComboBox class inherits from the UIComponent class. When accessing these properties from the ComboBox object, use the form *comboBoxInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the ComboBox class

The following table lists events of the ComboBox class.

Event	Description
ComboBox.change	Broadcast when the value of the combo box changes as a result of user interaction.
ComboBox.close	Broadcast when the list of the combo box begins to retract.
ComboBox.enter	Broadcast when the Enter key is pressed.
ComboBox.itemRollOut	Broadcast when the pointer rolls off a drop-down list item.
ComboBox.itemRollOver	Broadcast when a drop-down list item is rolled over.
ComboBox.open	Broadcast when the drop-down list begins to open.
ComboBox.scroll	Broadcast when the drop-down list is scrolled.

Events inherited from the UIObject class

The following table lists the events the ComboBox class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.

Event	Description
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the ComboBox class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

ComboBox.addItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
comboBoxInstance.addItem(label[, data])
comboBoxInstance.addItem({label:label[, data:data]})
comboBoxInstance.addItem(obj);
```

Parameters

label A string that indicates the label for the new item.

data The data for the item; it can be of any data type. This parameter is optional.

obj An object with a `label` property and an optional `data` property.

Returns

The index at which the item was added.

Description

Method; adds a new item to the end of the list.

Example

The following code adds an item to the `myComboBox` instance:

```
myComboBox.addItem("this is an Item");
```

ComboBox.addItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
comboBoxInstance.addItemAt(index, label[, data])
comboBoxInstance.addItemAt(index, {label:label[, data:data]})
comboBoxInstance.addItemAt(index, obj);
```

Parameters

index A number 0 or greater that indicates the position at which to insert the item (the index of the new item).

label A string that indicates the label for the new item.

data The data for the item; it can be of any data type. This parameter is optional.

obj An object with `label` and `data` properties.

Returns

The index at which the item was added.

Description

Method; adds a new item to the end of the list at the index specified by the *index* parameter. Indices greater than `ComboBox.length` are ignored.

Example

The following code inserts an item at index 3, which is the fourth position in the combo box list (0 is the first position):

```
myBox.addItemAt(3, "this is the fourth Item");
```

ComboBox.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(change){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  // your code here
}
comboBoxInstance.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners when the `ComboBox.selectedIndex` or `ComboBox.selectedItem` property changes as a result of user interaction.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox instance. The keyword `this`, used in an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ComboBox instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(change){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`comboBoxInstance`*) dispatches an event (in this case, `change`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call `addEventListener()` (see `EventDispatcher.addEventListener()`) on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example sends the instance name of the component that generated the `change` event to the Output panel:

```
form.change = function(eventObj){
  trace("Value changed to " + eventObj.target.value);
}
myCombo.addEventListener("change", form);
```

See also

`EventDispatcher.addEventListener()`

ComboBox.close()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
myComboBox.close()
```

Parameters

None.

Returns

Nothing.

Description

Method; closes the drop-down list.

Example

The following example closes the drop-down list of the `myBox` combo box:

```
myBox.close();
```

See also

`ComboBox.open()`

ComboBox.close

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(close){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.close = function(eventObject){
  // your code here
}
comboBoxInstance.addEventListener("close", listenerObject)
```

Description

Event; broadcast to all registered listeners when the drop-down list of the combo box is fully retracted.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox instance. The keyword `this`, used in an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ComboBox instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(close){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`comboBoxInstance`*) dispatches an event (in this case, `close`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example sends a message to the Output panel when the drop-down list begins to close:

```
form.close = function(){
  trace("The combo box has closed");
}
myCombo.addEventListener("close", form);
```

See also

`EventDispatcher.addEventListener()`

ComboBox.dataProvider

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`comboBoxInstance``.dataProvider`

Description

Property; the data model for items viewed in a list. The value of this property can be an array or any object that implements the DataProvider API. The default value is `[]`. The List component and the ComboBox component share the `dataProvider` property, and changes to this property are immediately available to both components.

The List component, like other data-aware components, adds methods to the Array object's prototype so that they conform to the DataProvider API (see DataProvider.as for details). Therefore, any array that exists at the same time as a list automatically has all the methods (`addItem()`, `getItemAt()`, and so on) needed for it to be the model of a list, and can be used to broadcast model changes to multiple components.

If the array contains objects, the `labelField` or `labelFunction` property is accessed to determine what parts of the item to display. The default value is `"label"`, so if such a field exists, it is chosen for display; if not, a comma-separated list of all fields is displayed.

Note: If the array contains strings at each index, and not objects, the list is not able to sort the items and maintain the selection state. Any sorting will cause the selection to be lost.

Any instance that implements the DataProvider API is eligible as a data provider for a List component. This includes Flash Remoting RecordSet objects, Firefly DataSet components, and so on.

Example

This example uses an array of strings to populate the drop-down list:

```
comboBox.dataProvider = ["Ground Shipping","2nd Day Air","Next Day Air"];
```

This example creates a data provider array and assigns it to the `dataProvider` property:

```
myDP = new Array();
list.dataProvider = myDP;

for (var i=0; i<accounts.length; i++) {
  // these changes to the DataProvider will be broadcast to the list
  myDP.addItem({label: accounts[i].name,
                data:  accounts[i].accountID});
}
```

ComboBox.dropdown

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox.dropdown

Description

Property (read-only); returns a reference to the list contained by the combo box. The List subcomponent isn't instantiated in the combo box until it needs to be displayed. However, when you access the `dropdown` property, the list is created.

See also

`ComboBox.dropdownWidth`

ComboBox.dropdownWidth

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`myComboBox``.dropdownWidth`

Description

Property; the width limit of the drop-down list, in pixels. The default value is the width of the ComboBox component (the TextInput instance plus the SimpleButton instance).

Example

The following code sets `dropdownWidth` to 150 pixels:

```
myComboBox.dropdownWidth = 150;
```

See also

`ComboBox.dropdown`

ComboBox.editable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`myComboBox``.editable`

Description

Property; indicates whether the combo box is editable (`true`) or not (`false`). In an editable combo box, a user can enter values into the text box that do not appear in the drop-down list. If a combo box is not editable, you cannot enter text into the text box. The text box displays the text of the item in the list. The default value is `false`.

Making a combo box editable clears the combo box text field. It also sets the selected index (and item) to `undefined`. To make a combo box editable and still retain the selected item, use the following code:

```
var ix = myComboBox.selectedIndex;
myComboBox.editable = true; // clears the text field
myComboBox.selectedIndex = ix; // copies the label back into the text field
```

Example

The following code makes `myComboBox` editable:

```
myComboBox.editable = true;
```

ComboBox.enter

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(enter){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.enter = function(eventObject){
  // your code here
}
comboBoxInstance.addEventListener("enter", listenerObject)
```

Description

Event; broadcast to all registered listeners when the user presses the Enter key in the text box. This event is a TextInput event that is broadcast only from editable combo boxes. For more information, see `TextInput.enter`.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox instance. The keyword `this`, used in an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ComboBox instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(enter){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*comboBoxInstance*) dispatches an event (in this case, `enter`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example sends a message to the Output panel when the drop-down list begins to close:

```
form.enter = function(){
  trace("The combo box enter event was triggered");
}
myCombo.addEventListener("enter", form);
```

See also

`EventDispatcher.addEventListener()`

ComboBox.getItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

comboBoxInstance`.getItemAt(`*index*`)`

Parameters

index The index of the item to retrieve. The index must be a number greater than or equal to 0, and less than the value of `ComboBox.length`.

Returns

The indexed item object or value. The value is undefined if the index is out of range.

Description

Method; retrieves the item at a specified index.

Example

The following code displays the item at index position 4:

```
trace(myBox.getItemAt(4).label);
```

ComboBox.itemRollOut

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(itemRollOut){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.itemRollOut = function(eventObject){
  // your code here
}
comboBoxInstance.addEventListener("itemRollOut", listenerObject)
```

Event object

In addition to the standard properties of the event object, the `itemRollOut` event has an `index` property. The index is the number of the item that the pointer rolled out of.

Description

Event; broadcast to all registered listeners when the pointer rolls out of drop-down list items. This is a List event that is broadcast from a combo box. For more information, see `List.itemRollOut`.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ComboBox instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(itemRollOut){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*comboBoxInstance*) dispatches an event (in this case, `itemRollOut`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Finally, you call the `addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

Example

The following example sends a message to the Output panel that indicates the index of the item that the pointer rolled out of:

```
form.itemRollOut = function (eventObj) {
  trace("Item #" + eventObj.index + " has been rolled out of.");
}
myCombo.addEventListener("itemRollOut", form);
```

See also

`ComboBox.itemRollOver`, `EventDispatcher.addEventListener()`

ComboBox.itemRollOver

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(itemRollOver){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.itemRollOver = function(eventObject){
  // your code here
}
comboBoxInstance.addEventListener("itemRollOver", listenerObject)
```

Event object

In addition to the standard properties of the event object, the `itemRollOver` event has an `index` property. The index is the number of the item that the pointer rolled over.

Description

Event; broadcast to all registered listeners when the mouse pointer rolls over drop-down list items. This is a List event that is broadcast from a combo box. For more information, see `List.itemRollOver`.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox instance. The keyword `this`, used in an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ComboBox instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(itemRollOver){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`comboBoxInstance`*) dispatches an event (in this case, `itemRollOver`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Finally, you call the `addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

Example

The following example sends a message to the Output panel that indicates the index of the item that the pointer rolled over:

```
form.itemRollOver = function (eventObj) {
  trace("Item #" + eventObj.index + " has been rolled over.");
}
myCombo.addEventListener("itemRollOver", form);
```

See also

`ComboBox.itemRollOut`, `EventDispatcher.addEventListener()`

ComboBox.labelField

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`myComboBox``.labelField`

Description

Property; the name of the field in `dataProvider` array objects to use as the label field. This is a property of the List component that is available from a ComboBox component instance. For more information, see `List.labelField`.

The default value is `undefined`.

Example

The following example sets the `dataProvider` property to an array of strings and sets the `labelField` property to indicate that the `name` field should be used as the label for the drop-down list:

```
myComboBox.dataProvider = [
  {name:"Gary", gender:"male"},
  {name:"Susan", gender:"female"} ];

myComboBox.labelField = "name";
```

See also

`List.labelFunction`

ComboBox.labelFunction

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox.labelFunction

Description

Property; a function that computes the label of a data provider item. You must define the function. The default value is `undefined`.

Example

The following example creates a data provider and then defines a function to specify what to use as the label in the drop-down list:

```
myComboBox.dataProvider  = [
  {firstName:"Nigel", lastName:"Pegg", age:"really young"},
  {firstName:"Gary", lastName:"Grossman", age:"young"},
  {firstName:"Chris", lastName:"Walcott", age:"old"},
  {firstName:"Greg", lastName:"Yachuk", age:"really old"} ];

myComboBox.labelFunction = function(itemObj){
  return (itemObj.lastName + ", " + itemObj.firstName);
}
```

See also

`List.labelField`

ComboBox.length

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox`.length`

Description

Property (read-only); the length of the drop-down list. This is a property of the List component that is available from a ComboBox instance. For more information, see `List.length`. The default value is 0.

Example

The following example stores the value of `length` to a variable:

```
dropdownItemCount = myBox.length;
```

ComboBox.open()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox`.open()`

Parameters

None.

Returns

Nothing.

Description

Method; opens the drop-down list.

Example

The following code opens the drop-down list for the `combo1` instance:

```
combo1.open();
```

See also

ComboBox.close()

ComboBox.open

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(open){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.open = function(eventObject){
  // your code here
}
comboBoxInstance.addEventListener("open", listenerObject)
```

Description

Event; broadcast to all registered listeners when the drop-down list is completely open.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox instance. The keyword `this`, used in an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ComboBox instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(open){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`comboBoxInstance`*) dispatches an event (in this case, `open`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Finally, you call the `addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

Example

The following example sends a message to the Output panel:

```
function open(evt) {
  trace("The combo box has opened with text " + evt.target.text);
}
myBox.addEventListener("open", this);
```

See also

ComboBox.close, EventDispatcher.addEventListener()

ComboBox.removeAll()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

comboBoxInstance.removeAll()

Parameters

None.

Returns

Nothing.

Description

Method; removes all items in the list. This is a method of the List component that is available from an instance of the ComboBox component.

Example

The following code clears the list:

```
myCombo.removeAll();
```

See also

ComboBox.removeItemAt(), ComboBox.replaceItemAt()

ComboBox.removeItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.removeItemAt(index)
```

Parameters

index A number that indicates the position of the item to remove. The index is zero-based.

Returns

An object; the removed item (undefined if no item exists).

Description

Method; removes the item at the specified index position. The list indices after the index indicated by the *index* parameter collapse by one. This is a method of the List component that is available from an instance of the ComboBox component.

Example

The following code removes the item at index position 3:

```
myCombo.removeItemAt(3);
```

See also

ComboBox.removeAll(), ComboBox.replaceItemAt()

ComboBox.replaceItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
comboBoxInstance.replaceItemAt(index, label[, data])
comboBoxInstance.replaceItemAt(index, {label:label[, data:data]})
comboBoxInstance.replaceItemAt(index, obj);
```

Parameters

index A number 0 or greater that indicates the position at which to insert the item (the index of the new item).

label A string that indicates the label for the new item.

data The data for the item. This parameter is optional.

obj An object with `label` and `data` properties.

Returns

Nothing.

Description

Method; replaces the content of the item at the specified index. This is a method of the List component that is available from the ComboBox component.

Example

The following example changes the third index position:

```
myCombo.replaceItemAt(3, "new label");
```

See also

ComboBox.removeAll(), ComboBox.removeItemAt()

ComboBox.restrict

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

comboBoxInstance.restrict

Description

Property; indicates the set of characters that a user can enter in the text field of a combo box. The default value is `undefined`. If this property is `null` or an empty string (`""`), a user can enter any character. If this property is a string of characters, the user can enter only characters in the string; the string is scanned from left to right. You can specify a range by using a dash (`-`).

If the string begins with a caret (`^`), all characters that follow the caret are considered unacceptable characters. If the string does not begin with a caret, the characters in the string are considered acceptable.

You can use the backslash (`\`) to enter a hyphen (`-`), caret (`^`), or backslash (`\`) character, as shown here:

```
\^
\-
\\
```

When you enter a backslash in the Actions panel within double quotation marks, it has a special meaning for the Actions panel's double-quote interpreter. It signifies that the character following the backslash should be treated "as is." For example, you could use the following code to enter a single quotation mark:

```
var leftQuote = "\'";
```

The Actions panel's restrict interpreter also uses the backslash as an escape character. Therefore, you may think that the following should work:

```
myText.restrict = "0-9\-\^\\";
```

However, since this expression is surrounded by double quotation marks, the value `0-9-^\` is sent to the restrict interpreter, and the restrict interpreter doesn't understand this value.

Because you must enter this expression within double quotation marks, you must not only provide the expression for the restrict interpreter, but you must also escape the expression so that it will be read correctly by the Actions panel's built-in interpreter for double quotation marks. To send the value `0-9\-\^\\` to the restrict interpreter, you must enter the following code:

```
myCombo.restrict = "0-9\\-\\^\\\\";
```

The `restrict` property restricts only user interaction; a script may put any text into the text field. This property does not synchronize with the Embed Font Outlines check boxes in the Property inspector.

Example

In the following example, the first line of code limits the text field to uppercase letters, numbers, and spaces. The second line of code allows all characters except lowercase letters.

```
my_combo.restrict = "A-Z 0-9";
my_combo.restrict = "^a-z";
```

The following code allows a user to enter the characters "0 1 2 3 4 5 6 7 8 9 - ^ \" in the instance `myCombo`. You must use a double backslash to escape the characters `-`, `^`, and `\`. The first `\` escapes the double quotation marks, and the second `\` tells the interpreter that the next character should not be treated as a special character.

```
myCombo.restrict = "0-9\\-\\^\\\\";
```

ComboBox.rowCount

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox.rowCount

Description

Property; the maximum number of rows visible in the drop-down list. The default value is 5.

If the number of items in the drop-down list is greater than the `rowCount` property, the list resizes and a scroll bar is displayed if necessary. If the drop-down list contains fewer items than the `rowCount` property, it resizes to the number of items in the list.

This behavior differs from the List component, which always shows the number of rows specified by its `rowCount` property, even if some empty space is shown.

If the value is negative or fractional, the behavior is undefined.

Example

The following example specifies that the combo box should have 20 or fewer rows visible:

```
myComboBox.rowCount = 20;
```

ComboBox.scroll

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(scroll){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.scroll = function(eventObject){
  // your code here
}
comboBoxInstance.addEventListener("scroll", listenerObject)
```

Event object

Along with the standard event object properties, the scroll event has one additional property, `direction`. It is a string with two possible values, `"horizontal"` or `"vertical"`. For a ComboBox `scroll` event, the value is always `"vertical"`.

Description

Event; broadcast to all registered listeners when the drop-down list is scrolled. This is a List component event that is available to the ComboBox component.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox instance. The keyword `this`, used in an `on()` handler attached to a component, refers to the instance. For example, the following code, attached to the ComboBox component instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(scroll){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*comboBoxInstance*) dispatches an event (in this case, `scroll`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Finally, you call the `addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

Example

The following example sends a message to the Output panel that indicates the index of the item that the list scrolled to:

```
form.scroll = function (eventObj) {
  trace("The list had been scrolled to item # " + eventObj.target.vPosition);
}
myCombo.addEventListener("scroll", form);
```

See also

`EventDispatcher.addEventListener()`

ComboBox.selectedIndex

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox`.selectedIndex`

Description

Property; the index number of the selected item in the drop-down list. The default value is 0. Assigning this property clears the current selection, selects the indicated item, and displays the label of that item in the combo box's text box.

If you assign an out-of-range value to this property, Flash ignores it. Entering text into the text field of an editable combo box sets `selectedIndex` to `undefined`.

Example

The following code selects the last item in the list:

```
myComboBox.selectedIndex = myComboBox.length-1;
```

See also

ComboBox.selectedItem

ComboBox.selectedItem

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox.selectedItem

Description

Property; the value of the selected item in the drop-down list.

If the combo box is editable, selectedItem returns undefined if the user enters any text in the text box. The property only has a value if you select an item from the drop-down list or set the value using ActionScript. If the combo box is static, the value of selectedItem is always valid; it returns undefined if there are no items in the list.

Example

The following example shows selectedItem if the data provider contains primitive types:

```
var item = myComboBox.selectedItem;
trace("You selected the item " + item);
```

The following example shows selectedItem if the data provider contains objects with label and data properties:

```
var obj = myComboBox.selectedItem;
trace("You have selected the color named: " + obj.label);
trace("The hex value of this color is: " + obj.data);
```

See also

ComboBox.dataProvider, ComboBox.selectedIndex

ComboBox.sortItems()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myComboBox.sortItems([*compareFunc*], [*optionsFlag*])

Parameters

compareFunc A reference to a function that compares two items to determine their sort order. For details, see `Array.sort()` in *Flash ActionScript Language Reference*. This parameter is optional.

optionsFlag Lets you perform multiple sorts of different types on a single array without having to replicate the entire array or re-sort it repeatedly. This parameter is optional.

The following are possible values for *optionsFlag*:

- `Array.DESCENDING`, which sorts highest to lowest.
- `Array.CASEINSENSITIVE`, which sorts without regard to case.
- `Array.NUMERIC`, which sorts numerically if the two elements being compared are numbers. If they aren't numbers, use a string comparison (which can be case-insensitive if that flag is specified).
- `Array.UNIQUESORT`, which returns an error code (0) instead of a sorted array if two objects in the array are identical or have identical sort fields.
- `Array.RETURNINDEXEDARRAY`, which returns an integer index array that is the result of the sort. For example, the following array would return the second line of code and the array would remain unchanged:

```
["a", "d", "c", "b"]
[0, 3, 2, 1]
```

You can combine these options into one value. For example, the following code combines options 3 and 1:

```
array.sort (Array.NUMERIC | Array.DESCENDING)
```

Returns

Nothing.

Description

Method; sorts the items in the combo box according to the specified compare function or according to the specified sort options.

Example

This example sorts according to uppercase labels. The items `a` and `b` are passed to the function and contain `label` and `data` fields:

```
myComboBox.sortItems(upperCaseFunc);
function upperCaseFunc(a,b){
    return a.label.toUpperCase() > b.label.toUpperCase();
}
```

The following example uses the `upperCaseFunc()` function defined above, along with the *optionsFlag* parameter to sort the elements of a ComboBox instance named `myComboBox`:

```
myComboBox.addItem("Mercury");
myComboBox.addItem("Venus");
myComboBox.addItem("Earth");
myComboBox.addItem("planet");
```

```
myComboBox.sortItems(upperCaseFunc, Array.DESCENDING);
// The resulting sort order of myComboBox will be:
// Venus
// planet
// Mercury
// Earth
```

ComboBox.sortItemsBy()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myComboBox.sortItemsBy(*fieldName*, *order* [*optionsFlag*])

Parameters

fieldName A string that specifies the name of the field to use for sorting. This value is usually `"label"` or `"data"`.

order A string that specifies whether to sort the items in ascending order (`"ASC"`) or descending order (`"DESC"`).

optionsFlag Lets you perform multiple sorts of different types on a single array without having to replicate the entire array or re-sort it repeatedly. This parameter is optional, but if used, should replace the *order* parameter.

The following are possible values for *optionsFlag*:

- `Array.DESCENDING`, which sorts highest to lowest.
- `Array.CASEINSENSITIVE`, which sorts without regard to case.
- `Array.NUMERIC`, which sorts numerically if the two elements being compared are numbers. If they aren't numbers, use a string comparison (which can be case-insensitive if that flag is specified).
- `Array.UNIQUESORT`, which returns an error code (0) instead of a sorted array if two objects in the array are identical or have identical sort fields.
- `Array.RETURNINDEXEDARRAY`, which returns an integer index array that is the result of the sort. For example, the following array would return the second line of code and the array would remain unchanged:

  ```
  ["a", "d", "c", "b"]
  [0, 3, 2, 1]
  ```

You can combine these options into one value. For example, the following code combines options 3 and 1:

```
array.sort (Array.NUMERIC | Array.DESCENDING)
```

Returns

Nothing.

Description

Method; sorts the items in the combo box alphabetically or numerically, in the specified order, using the specified field name. If the *fieldName* items are a combination of text strings and integers, the integer items are listed first. The *fieldName* parameter is usually `"label"` or `"data"`, but advanced programmers may specify any primitive value. If you want, you can use the *optionsFlag* parameter to specify a sorting style.

Example

The following examples are based on a ComboBox instance named `myComboBox`, which contains four elements labeled `"Apples"`, `"Bananas"`, `"cherries"`, and `"Grapes"`:

```
// First, populate the ComboBox with the elements.
myComboBox.addItem("Bananas");
myComboBox.addItem("Apples");
myComboBox.addItem("cherries");
myComboBox.addItem("Grapes");

// The following statement sorts using the order parameter set to "ASC",
// and results in a sort that places "cherries" at the bottom of the list
// because the sort is case-sensitive.
myDP.sortItemsBy("label", "ASC");
// resulting order: Apples, Bananas, Grapes, cherries

// The following statement sorts using the order parameter set to "DESC",
// and results in a sort that places "cherries" at the top of the list
// because the sort is case-sensitive.
myComboBox.sortItemsBy("label", "DESC");
// resulting order: cherries, Grapes, Bananas, Apples

// The following statement sorts using the optionsFlag parameter set to
// Array.CASEINSENSITIVE. Note that an ascending sort is the default setting.
myComboBox.sortItemsBy("label", Array.CASEINSENSITIVE);
// resulting order: Apples, Bananas, cherries, Grapes

// The following statement sorts using the optionsFlag parameter set to
// Array.CASEINSENSITIVE | Array.DESCENDING.
myComboBox.sortItemsBy("label", Array.CASEINSENSITIVE | Array.DESCENDING);
// resulting order: Grapes, cherries, Bananas, Apples
```

ComboBox.text

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox.text

Description

Property; the text of the text box. You can get and set this value for editable combo boxes. For static combo boxes, the value is read-only.

Example

The following example sets the current `text` value of an editable combo box:

```
myComboBox.text = "California";
```

ComboBox.textField

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox.textField

Description

Property (read-only); a reference to the TextInput component contained by the ComboBox component.

This property lets you access the underlying TextInput component so that you can manipulate it. For example, you might want to change the selection of the text box or restrict the characters that can be entered in it.

Example

The following code restricts the text box of `myComboBox` so that it only accept numbers:

```
myComboBox.textField.restrict = "0-9";
```

ComboBox.value

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myComboBox.value

Description

Property (read-only); if the combo box is editable, `value` returns the value of the text box. If the combo box is static, `value` returns the value of the drop-down list. The value of the drop-down list is the `data` field, or, if the `data` field doesn't exist, the `label` field.

Example

The following example puts the data into the combo box by setting the `dataProvider` property. It then displays the `value` in the Output panel. Finally, it selects `"California"` and displays it in the text box.

```
cb.dataProvider = [
  {label:"Alaska", data:"AZ"},
  {label:"California", data:"CA"},
  {label:"Washington", data:"WA"}];
cb.editable = true;
cb.selectedIndex = 1;
trace('Editable value is "California": '+ cb.value);
cb.editable = false;
cb.selectedIndex = 1;
trace('Non-editable value is "CA": '+ cb.value);
```

Data binding classes (Flash Professional only)

The data binding classes provide the runtime functionality for the data binding feature in Flash MX Professional 2004. You can visually create and configure data bindings in the Flash authoring environment by using the Bindings tab in the Component inspector, or you can programmatically create and configure bindings by using the classes in the mx.data.binding package.

For an overview of data binding and how to visually create data bindings in the Flash authoring tool, see "Data binding (Flash Professional only)" in *Using Flash*.

Making data binding classes available at runtime (Flash Professional only)

To compile your SWF file, your library must contain SWC files that contain the byte code for the data binding classes and web service classes. If you create data bindings in Flash while authoring, the relevant component classes are automatically added to the library. If you work with data binding and web services at runtime, you must add the classes to your FLA file's library. You can get these SWC files from the Classes common library.

To add the SWC files to your library:

1. Select the Classes library (Window > Other Panels > Common Libraries > Classes).
2. Open the library for your document (Window > Library).
3. Drag the appropriate SWC files (DataBindingClasses, WebServiceClasses, or both) from the Classes library into your document's library.

For more information on these classes, see "Binding class (Flash Professional only)" on page 214 and "Web service classes (Flash Professional only)" on page 842.

Classes in the mx.data.binding package (Flash Professional only)

The following table lists the classes in the mx.data.binding package:

Class	Description
Binding class (Flash Professional only)	Creates a binding between two endpoints.
ComponentMixins class (Flash Professional only)	Adds data binding functionality to components.
CustomFormatter class (Flash Professional only)	The base class for creating custom formatter classes.
CustomValidator class (Flash Professional only)	The base class for creating custom validator classes.
DataType class (Flash Professional only)	Provides read and write access to data fields of a component property.

Class	Description
EndPoint class (Flash Professional only)	Defines the source or destination of a binding.
TypedValue class (Flash Professional only)	Contains a data value and information about the value's data type.

Binding class (Flash Professional only)

ActionScript Class Name mx.data.binding.Binding

The Binding class defines an association between two endpoints, a source and a destination. It listens for changes to the source endpoint and copies the changed data to the destination endpoint each time the source changes.

You can write custom bindings by using the Binding class (and supporting classes), or use the Bindings tab in the Component inspector.

Note: To make this class available at runtime, you must include the data binding classes in your FLA document. For more information, see "Making data binding classes available at runtime (Flash Professional only)" on page 213.

For an overview of the classes in the mx.data.binding package, see "Classes in the mx.data.binding package (Flash Professional only)" on page 213.

Method summary for the Binding class

The following table lists the methods of the Binding class.

Method	Description
`Binding.execute()`	Fetches the data from the source component, formats it, and assigns it to the destination component.

Constructor for the Binding class

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
new Binding(source, destination, [format], [isTwoWay])
```

Parameters

source A source endpoint of the binding. This parameter is nominally of type mx.data.binding.EndPoint, but can be any ActionScript object that has the required Endpoint fields (see "EndPoint class (Flash Professional only)" on page 224).

destination The destination endpoint of the binding. This parameter is nominally of type `mx.data.binding.EndPoint`, but can be any ActionScript object that has the required Endpoint fields.

format An optional object that contains formatting information. The object must have the following properties:

- `cls` An ActionScript class that extends the class mx.data.binding.DataAccessor.
- `settings` An object whose properties provide optional settings for the formatter class specified by `cls`.

isTwoWay An optional Boolean value that specifies whether the new Binding object is bidirectional (`true`) or not (`false`). The default value is `false`.

Returns

Nothing.

Description

Constructor; creates a new Binding object. You can bind data to any ActionScript object that has properties and emits events including, but not limited to, components.

A binding object exists as long as the innermost movie clip contains both the source and destination components. For example, if movie clip named A contains components X and Y, and there is a binding between X and Y, then the binding is in effect as long as movie clip A exists.

Note: It's not necessary to retain a reference to the new Binding object. As soon as the Binding object is created, it immediately begins listening for "changed" events emitted by either endpoint. In some cases, however, you might want to save a reference to the new Binding object, so that you can call its `execute()` method at a later time (see `Binding.execute()`).

Example

In this example, the `text` property of a TextInput component (`src_txt`) is bound to the `text` property of another TextInput component (`dest_txt`). When the `src_txt` text field loses focus (that is, when the `focusOut` event is generated), the value of its `text` property is copied into `dest_txt.text`.

```
import mx.data.binding.*;
var src = new EndPoint();
src.component = src_txt;
src.property = "text";
src.event = "focusOut";

var dest= new EndPoint();
dest.component = dest_txt;
dest.property = "text";

new Binding(src, dest);
```

The following example demonstrates how to create a Binding object that uses a custom formatter class. For more information, see "CustomFormatter class (Flash Professional only)" on page 217.

```
import mx.data.binding.*;
var src = new EndPoint();
```

```
src.component = src_txt;
src.property = "text";
src.event = "focusOut";

var dest= new EndPoint();
dest.component = text_dest;
dest.property = "text";

new Binding(src, dest, {cls: mx.data.formatters.Custom, settings: {classname:
  "com.mycompany.SpecialFormatter"}});
```

Binding.execute()

Availability

Flash Player 6.

Edition

Flash MX Professional 2004.

Usage

myBinding.execute([*reverse*])

Parameters

reverse A Boolean value that specifies whether the binding should also be executed from the destination to the source (`true`), or only from the source to the destination (`false`). By default, this value is `false`.

Returns

A `null` value if the binding executed successfully; otherwise, the method returns an array of error message strings that describe the errors that prevented the binding from executing.

Description

Method; fetches the data from the source component and assigns it to the destination component. If the binding uses a formatter, then the data is formatted before being assigned to the destination.

This method also validates the data and causes either a `valid` or `invalid` event to be emitted by the destination and source components. Data is assigned to the destination even if it's invalid, unless the destination is read-only.

If the *reverse* parameter is set to `true` and the binding is two-way, then the binding is executed in reverse (from the destination to the source).

Example

The following code, attached to a Button component instance, executes the binding in reverse (from the destination component to the source component) when the button is clicked.

```
on(click) {
  _root.myBinding.execute(true);
}
```

CustomFormatter class (Flash Professional only)

ActionScript Class Name mx.data.binding.CustomFormatter

The CustomFormatter class defines two methods, `format()` and `unformat()`, that provide the ability to transform data values from a specific data type to String, and vice versa. By default, these methods do nothing; you must implement them in a subclass of mx.data.binding.CustomFormatter.

To create your own custom formatter, you first create a subclass of CustomFormatter that implements `format()` and `unformat()` methods. You can then assign that class to a binding between components either by creating a new Binding object with ActionScript (see "Binding class (Flash Professional only)" on page 214), or by using the Bindings tab in the Component inspector. For information on assigning a formatter class using the Component inspector, see "Schema formatters" in *Using Flash*.

You can also assign a formatter class to a component property on the Component inspector's Schema tab. However, in that case, the formatter will be used only when the data is needed in the form of a string. In contrast, formatters assigned with the Bindings panel, or created with ActionScript, are used whenever when the binding is executed.

For an example of writing and assigning a custom formatter using ActionScript, see "Sample custom formatter" on page 217.

Note: To make this class available at runtime, you must include the data binding classes in your FLA document.

For an overview of the classes in the mx.data.binding package, see "Classes in the mx.data.binding package (Flash Professional only)" on page 213.

Sample custom formatter

The following example demonstrates how to create a custom formatter class and then apply it to a binding between two components by using ActionScript. In this example, the current value of a NumericStepper component (its `value` property) is bound to the current value of a TextInput component (its `text` property). The custom formatter class formats the current numeric value of the NumericStepper component (for example, 1, 2, or 3) as its English word equivalent (for example, "one", "two", or "three") before assigning it to the TextInput component.

To create and use a custom formatter:

1. In Flash MX Professional 2004, create a new ActionScript file.
2. Add the following code to the file:

```
// NumberFormatter.as
class NumberFormatter extends mx.data.binding.CustomFormatter {
  // Format a Number, return a String
  function format(rawValue) {
    var returnValue;
    var strArray = new Array('one', 'two', 'three');
    var numArray = new Array(1, 2, 3);
    returnValue = 0;
    for (var i = 0; i<strArray.length; i++) {
```

```
        if (rawValue == numArray[i]) {
          returnValue = strArray[i];
          break;
        }
      }
      return returnValue;
    } // convert a formatted value, return a raw value
    function unformat(formattedValue) {
      var returnValue;
      var strArray = new Array('one', 'two', 'three');
      var numArray = new Array(1, 2, 3);
      returnValue = "invalid";
      for (var i = 0; i<strArray.length; i++) {
        if (formattedValue == strArray[i]) {
          returnValue = numArray[i];
          break;
        }
      }
      return returnValue;
    }
}
```

3. Save the ActionScript file as NumberFormatter.as.
4. Create a new Flash (FLA) document.
5. From the Components panel, drag a TextInput component to the Stage and name it **textInput**. Then drag a NumericStepper component to the Stage and name it **stepper**.
6. Open the Timeline and select the first frame on Layer 1.
7. In the Actions panel, add the following code to the Actions panel:

```
import mx.data.binding.*;
var x:NumberFormatter;
var customBinding = new Binding({component:stepper, property:"value",
  event:"change"}, {component:textInput, property:"text",
  event:"enter,change"}, {cls:mx.data.formatters.Custom,
  settings:{classname:"NumberFormatter"}});
```

   The second line of code (`var x:NumberFormatter`) ensures that the byte code for your custom formatter class is included in the compiled SWF file.

8. Select Window > Panels > Other Panels > Classes to open the Classes library.
9. Open your document's library by selecting Window > Library.
10. Drag DataBindingClasses from the Classes library to your document's library.

    This makes the data binding runtime classes available for your document.

11. Save the FLA file to the same folder that contains NumberFormatter.as.
12. Test the file (Control > Test Movie).

    Click the buttons on the NumericStepper component and watch the contents of the TextInput component update.

Method summary for the CustomFormatter class

The following table lists the methods of the CustomFormatter class.

Method	Description
`CustomFormatter.format()`	Converts from a raw data type to a new object.
`CustomFormatter.unformat()`	Converts from a string, or other data type, to a raw data type.

CustomFormatter.format()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

This method is called automatically; you don't invoke it directly.

Parameters

`rawData` The data to be formatted.

Returns

A formatted value.

Description

Method; converts from a raw data type to a new object.

This method is not implemented by default. You must define it in your subclass of mx.data.binding.CustomFormatter.

For more information, see "Sample custom formatter" on page 217.

CustomFormatter.unformat()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

This method is called automatically; you don't invoke it directly.

Parameters

`formattedData` The formatted data to convert back to the raw data type.

Returns

An unformatted value.

Description

Method; converts from a string, or other data type, to the raw data type. This transformation should be the exact inverse transformation of `CustomFormatter.format()`.

This method is not implemented by default. You must define it in your subclass of mx.data.binding.CustomFormatter.

For more information, see "Sample custom formatter" on page 217.

CustomValidator class (Flash Professional only)

ActionScript Class Name mx.data.binding.CustomValidator

You use the CustomValidator class when you want to perform custom validation of a data field contained by a component.

To create a custom validator class, you first create a subclass of mx.data.binding.CustomValidator that implements a method named `validate()`. This method is automatically passed a value to be validated. For more information about how to implement this method, see `CustomValidator.validate()`.

Next, you assign your custom validator class to a field of a component by using the Component inspector's Schema tab. For an example of creating and using a custom validator class, see the Example section in the `CustomValidator.validate()` entry.

To assign a custom validator:

1. In the Component inspector, select the Schema tab.
2. Select the field you want to validate, and then select Custom from the Data Type pop-up menu.
3. Select the Validation Options field (at the bottom of the Schema tab), and click the magnifying glass icon to open the Custom Validation Settings dialog box.
4. In the ActionScript Class text box, enter the name of the custom validator class you created.

   In order for the class you specify to be included in the published SWF file, it must be in the classpath.

Note: To make this class available at runtime, you must include the data binding classes in your FLA document. .

For an overview of the classes in the mx.data.binding package, see "Classes in the mx.data.binding package (Flash Professional only)" on page 213.

Method summary for the CustomValidator class

The following table lists the methods of the CustomValidator class.

Method	Description
`CustomValidator.validate()`	Performs validation on data.
`CustomValidator.validationError()`	Reports validation errors.

CustomValidator.validate()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

This method is called automatically; you don't invoke it directly.

Parameters

`value` The data to be validated; it can be of any type.

Returns

Nothing.

Description

Method; called automatically to validate the data contained by the *`value`* parameter. You must implement this method in your subclass of CustomValidator; the default implementation does nothing.

You can use any ActionScript code to examine and validate the data. If the data is not valid, this method should call `this.validationError()` with an appropriate message. You can call `this.validationError()` more than once if there are several validation problems with the data.

Since `validate()` might be called repeatedly, avoid adding code that takes a long time to complete. Your implementation of this method should only check for validity, and then report any errors using `CustomValidator.validationError()`. Similarly, your implementation should not take any action as a result of the validation test, such as alerting the end user. Instead, create event listeners for `valid` and `invalid` events and alert the end user from those event listeners (see the example below).

Example

The following procedure demonstrates how to create and use a custom validator class. The `validate()` method of the CustomValidator class OddNumbersOnly.as determines any value that is not an odd number to be invalid. The validation occurs whenever a change occurs in the value of a NumericStepper component, which is bound to the `text` property of a Label component.

To create and use a custom validator class:

1. In Flash MX Professional 2004, create a new ActionScript (AS) file.
2. Add the following code to the AS file:

```
class OddNumbersOnly extends mx.data.binding.CustomValidator
{
  public function validate(value) {
    // make sure the value is of type Number
    var n = Number(value);
    if (String(n) == "NaN") {
      this.validationError("'" + value + "' is not a number.");
      return;
    }
    // make sure the number is odd
    if (n % 2 == 0) {
      this.validationError("'" + value + "' is not an odd number.");
      return;
    }
    // data is OK, no need to do anything, just return
  }
}
```

3. Save the AS file as OddNumbersOnly.as.

   Note: The name of the AS file must match the name of the class.

4. Create a new Flash (FLA) document.
5. Open the Components panel.
6. Drag a NumericStepper component from the Components panel to the Stage and name it **stepper**.
7. Drag a Label component to the Stage and name it **textLabel**.
8. Drag a TextArea component to the Stage and name it **status**.
9. Select the NumericStepper component, and open the Component inspector.
10. Select the Bindings tab in the Component inspector, and click the Add Binding (+) button.
11. Select the Value property (the only one) in the Add Bindings dialog box, and click OK.
12. In the Component inspector, double-click Bound To in the Binding Attributes pane of the Bindings tab to open the Bound To dialog box.
13. In the Bound To dialog box, select the Label component in the Component Path pane and its `text` property in the Schema Location pane. Click OK.
14. Select the Label component on the Stage, and click the Schema tab in the Component Inspector panel.
15. In the Schema Attributes pane, select Custom from the Data Type pop-up menu.
16. Double-click the Validation Options field in the Schema Attributes pane to open the Custom Validation Settings dialog box.
17. In the ActionScript Class text box, enter **OddNumbersOnly**, which is the name of the ActionScript class you created previously. Click OK.

18. Open the Timeline and select the first frame on Layer 1.
19. Open the Actions panel.
20. Add the following code to the Actions panel:

```
function dataIsInvalid(evt) {
  if (evt.property == "text") {
    status.text = evt.messages;
  }
}

function dataIsValid(evt) {
  if (evt.property == "text") {
    status.text = "OK";
  }
}

textLabel.addEventListener("valid", dataIsValid);
textLabel.addEventListener("invalid", dataIsInvalid);
```

21. Save the FLA file as OddOnly.fla to the same folder that contains OddNumbersOnly.as.
22. Test the SWF file (Control > Test Movie).

    Click the arrows on the NumericStepper component to change its value. Notice the message that appears in the TextArea component when you choose even and odd numbers.

CustomValidator.validationError()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
this.validationError(errorMessage)
```

Note: This method can be invoked only from within a custom validator class; the keyword `this` refers to the current CustomValidator object.

Parameters

`errorMessage` A string that contains the error message to be reported.

Returns

Nothing.

Description

Method; called from the `validate()` method of your subclass of CustomValidator to report validation errors. If you don't call `validationError()`, a `valid` event is generated when `validate()` finishes executing. If you call `validationError()` one or more times from within the `validate()`, an `invalid` event is generated after `validate()` returns.

Each message you pass to `validationError()` is available in the `messages` property of the event object that was passed to the `invalid` event handler.

Example

See the Example section for `CustomValidator.validate()`.

EndPoint class (Flash Professional only)

ActionScript Class Name mx.data.binding.EndPoint

The EndPoint class defines the source or destination of a binding. EndPoint objects define a constant value, component property, or particular field of a component property, from which you can get data, or to which you can assign data. They can also define an event, or list of events, that a Binding object listens for; when the specified event occurs, the binding executes.

When you create a new binding with the Binding class constructor, you pass it two EndPoint objects: one for the source and one for the destination.

```
new mx.data.binding.Binding(srcEndPoint, destEndPoint);
```

The EndPoint objects, `srcEndPoint` and `destEndPoint`, might be defined as follows:

```
var srcEndPoint = new mx.data.binding.EndPoint();
var destEndPoint = new mx.data.binding.EndPoint();
srcEndPoint.component = source_txt;
srcEndPoint.property = "text";
srcEndPoint.event = "focusOut";
destEndPoint.component = dest_txt;
destEndPoint.property = "text";
```

In English, the above code means "When the source text field loses focus, copy the value of its `text` property into the `text` property of the destination text field."

You can also pass generic ActionScript objects to the Binding constructor, rather than passing explicitly constructed EndPoint objects. The only requirement is that the objects define the required EndPoint properties, `component` and `property`. The following code is equivalent to that shown above.

```
var srcEndPoint = {component:source_txt, property:"text"};
var destEndPoint = {component:dest_txt, property:"text"};
new mx.data.binding.Binding(srcEndPoint, destEndPoint);
```

Note: To make this class available at runtime, you must include the data binding classes in your FLA document.

For an overview of the classes in the mx.data.binding package, see "Classes in the mx.data.binding package (Flash Professional only)" on page 213.

Property summary for the EndPoint class

The following table lists the properties of the EndPoint class.

Method	Description
`EndPoint.component`	A reference to a component instance.
`EndPoint.constant`	A constant value.
`EndPoint.event`	The name of an event, or array of event names, that the component will emit when the data changes.
`EndPoint.location`	The location of a data field within the property of the component instance.
`EndPoint.property`	The name of a property of the component instance specified by `EndPoint.component`.

Constructor for the EndPoint class

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
new EndPoint()
```

Returns

Nothing.

Description

Constructor; creates a new EndPoint object.

Example

This example creates a new EndPoint object named `source_obj` and assigns values to its `component` and `property` properties:

```
var source_obj = new mx.data.binding.EndPoint();
source_obj.component = myTextField;
source_obj.property = "text";
```

EndPoint.component

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

endPointObj.component

Description

Property; a reference to a component instance.

Example

This example assigns an instance of the List component (`listBox1`) as the component parameter of an EndPoint object.

```
var sourceEndPoint = new mx.data.binding.EndPoint();
sourceEndPoint.component=listBox1;
```

EndPoint.constant

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

endPoint_src.constant

Description

Property; a constant value assigned to an EndPoint object. This property can be applied only to EndPoint objects that are the source, not the destination, of a binding between components. The value can be of any data type that is compatible with the destination of the binding. If this property is specified, all other EndPoint properties for the specified EndPoint object are ignored.

Example

In this example, the string constant value "hello" is assigned to an EndPoint object's constant property:

```
var sourceEndPoint = new mx.data.binding.EndPoint();
sourceEndPoint.constant="hello";
```

EndPoint.event

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

endPointObj.event

Description

Property; specifies the name of an event, or an array of event names, generated by the component when data assigned to the bound property changes. When the event occurs, the binding executes.

The specified event only applies to components that are used as the source of a binding, or as the destination of a two-way binding. For more information about creating two-way bindings, see "Binding class (Flash Professional only)" on page 214.

Example

In this example, the `text` property of one TextInput (`src_txt`) component is bound to the same property of another TextInput component (`dest_txt`). The binding is executed when either the `focusOut` or `enter` event is emitted by the `src_txt` component.

```
var source = {component:src_txt, property:"text", event:["focusOut",
  "enter"]};
var dest = {component:myTextArea, property:"text"};
var newBind = new mx.data.binding.Binding(source, dest);
```

EndPoint.location

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

endPointObj`.location`

Description

Property; specifies the location of a data field within the property of the component instance. There are four ways to specify a location: as a string that contains an XPath expression, as a string that contains an ActionScript path, as an array of strings, or as an object.

XPath expressions can only be used when the data is an XML object. (See Example 1 below.) For a list of supported XPath expressions, see "Adding bindings using path expressions" in *Using Flash*.

For XML and ActionScript objects, you can also specify a string that contains an ActionScript path. An ActionScript path contains the names of fields separated by dots (for example, `"a.b.c"`).

You can also specify an array of strings as a location. Each string in the array "drills down" another level of nesting. You can use this technique with both XML and ActionScript data. (See Example 2 below.) When used with ActionScript data, an array of strings is equivalent to use of an ActionScript path; that is, the array `["a","b","c"]` is equivalent to `"a.b.c"`.

If you specify an object as the location, the object must specify two properties: `path` and `indices`. The `path` property is an array of strings, as discussed above, except that one or more of the specified strings may be the special token `"[n]"`. For each occurrence of this token in `path`, there must be a corresponding index item in `indices`. As the path is evaluated, the indices are used to index into arrays. The index item can be any EndPoint object. This type of location can be applied to ActionScript data only—not XML. (See Example 3 below.)

Example

Example 1: This example uses an XPath expression to specify the location of a node named `zip` in an XML object:

```
var sourceEndPoint = new mx.databinding.EndPoint();
var sourcObj=new Object();
sourceObj.xml=new XML("<zip>94103</zip>");
sourceEndPoint.component=sourceObj;
sourceEndPoint.property="xml";
sourceEndPoint.location="/zip";//
```

Example 2: This example uses an array of strings to "drill down" to a nested movie clip property:

```
var sourceEndPoint = new mx.data.binding.EndPoint();
// assume movieClip1.ball.position exists
sourceEndPoint.component=movieClip1;
sourceEndPoint.property="ball";
// access movieClip1.ball.position.x
sourceEndPoint.location=["position","x"];
```

Example 3: This example shows how to use an object to specify the location of a data field in a complex data structure:

```
var city=new Object();
city.theaters = [{theater: "t1", movies: [{name: "Good,Bad,Ugly"},
  {name:"Matrix Reloaded"}]}, {theater: "t2", movies: [{name: "Gladiator"},
  {name: "Catch me if you can"}]}];
var srcEndPoint = new EndPoint();
srcEndPoint.component=city;
srcEndPoint.property="theaters";
srcEndPoint.location = {path: ["[n]","movies","[n]","name"], indices:
  [{constant:0},{constant:0}]};
```

EndPoint.property

Availability

Flash Player 6 (6.0 79.0)

Edition

Flash MX Professional 2004.

Usage

endPointObj.property

Description

Property; specifies a property name of the component instance specified by `EndPoint.component` that contains the bindable data.

Note: `EndPoint.component` and `EndPoint.property` must combine to form a valid ActionScript object/property combination.

Example

This example binds the `text` property of one TextInput component (`text_1`) to the same property in another TextInput component (`text_2`).

```
var sourceEndPoint = {component:text_1, property:"text"};
var destEndPoint = {component:text_2, property:"text"};
new Binding(sourceEndPoint, destEndPoint);
```

ComponentMixins class (Flash Professional only)

ActionScript Class Name mx.data.binding.ComponentMixins

The ComponentMixins class defines properties and methods that are automatically added to any object that is the source or destination of a binding, or to any component that's the target of a `ComponentMixins.initComponent()` method call. These properties and methods do not affect normal component functionality; rather, they add functionality that is useful with data binding.

Note: To make this class available at runtime, you must include the data binding classes in your FLA document.

For an overview of the classes in the mx.data.binding package, see "Classes in the mx.data.binding package (Flash Professional only)" on page 213.

Method summary for the ComponentMixins class

The following table lists the methods of the ComponentMixins class.

Method	Description
`ComponentMixins.getField()`	Returns an object for getting and setting the value of a field at a specific location in a component property.
`ComponentMixins.initComponent()`	Adds the ComponentMixins methods to a component.
`ComponentMixins.refreshDestinations()`	Executes all the bindings that have this object as the source endpoint.
`ComponentMixins.refreshFromSources()`	Executes all bindings that have this component as the destination endpoint.
`ComponentMixins.validateProperty()`	Checks to see if the data in the indicated property is valid.

ComponentMixins.getField()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.getField(propertyName, [location])
```

Parameters

propertyName A string that contains the name of a property of the specified component.

location An optional parameter that indicates the location of a field within the component property. This is useful if *propertyName* specifies a complex data structure and you are interested in a particular field of that structure. The *location* property can take one of three forms:

- A string that contains an XPath expression. This is only valid for XML data structures. For a list of supported XPath expressions, see "Adding bindings using path expressions" in *Using Flash*.
- A string that contains field names, separated by dots—for example, `"a.b.c"`. This form is permitted for any complex data (ActionScript or XML).
- An array of strings, where each string is a field name—for example, `["a", "b", "c"]`. This form is permitted for any complex data (ActionScript or XML).

Returns

A DataType object.

Description

Method; returns a DataType object whose methods you can use to get or set the data value in the component property at the specified field location. For more information, see "DataType class (Flash Professional only)" on page 234.

Example

This example uses the `DataType.setAsString()` method to set the value of a field located in a component's property. In this case the property (`results`) is a complex data structure.

```
import mx.data.binding.*;
var field : DataType = myComponent.getField("results", "po.address.name1");
field.setAsString("Teri Randall");
```

See also

`DataType.setAsString()`

ComponentMixins.initComponent()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
mx.data.binding.ComponentMixins.initComponent(componentInstance)
```

Parameters

componentInstance A reference to a component instance.

Returns

Nothing.

Description

Method (static); adds all the ComponentMixins methods to the component specified by *componentInstance*. This method is called automatically for all components involved in a data binding. To make the ComponentMixins methods available for a component that is not involved in a data binding, you must explicitly call this method for that component.

Example

The following code makes the ComponentMixins methods available to a DataSet component:

```
mx.data.binding.ComponentMixins.initComponent(_root.myDataSet);
```

ComponentMixins.refreshDestinations()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.refreshDestinations()
```

Parameters

None.

Returns

Nothing.

Description

Method; executes all the bindings for which *componentInstance* is the source EndPoint object. This method lets you execute bindings whose sources do not emit a “data changed” event.

Example

The following example executes all the bindings for which the DataSet component instance named `user_data` is the source EndPoint object:

```
user_data.refreshDestinations();
```

ComponentMixins.refreshFromSources()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.refreshFromSources()
```

Parameters

None.

Returns

Nothing.

Description

Method; executes all bindings for which *componentInstance* is the destination EndPoint object. This method lets you execute bindings that have constant sources, or sources that do not emit a "data changed" event.

Example

The following example executes all the bindings for which the ListBox component instance named `cityList` is the destination EndPoint object:

```
cityList.refreshFromSources();
```

ComponentMixins.validateProperty()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.validateProperty(propertyName)
```

Parameters

propertyName A string that contains the name of a property that belongs to *componentInstance*.

Returns

An array, or `null`.

Description

Method; determines if the data in *`propertyName`* is valid based on the property's schema settings. The property's schema settings are those specified on the Schema tab in the Component Inspector panel.

The method returns `null` if the data is valid; otherwise, it returns an array of error messages as strings.

Validation applies only to fields that have schema information available. If a field is an object that contains other fields, each "child" field is validated, and so on, recursively. Each individual field dispatches a `valid` or `invalid` event, as necessary. For each data field contained by *`propertyName`*, this method dispatches `valid` or `invalid` events, as follows:

- If the value of the field is `null`, and is *not* required, the method returns `null`. No events are generated.
- If the value the field is `null`, and *is* required, an error is returned and an `invalid` event is generated.
- If the value of the field is not `null` and the field's schema does *not* have a validator, the method returns `null`; no events are generated.
- If the value is not `null` and the field's schema *does* define a validator, the data is processed by the validator object. If the data is valid, a `valid` event is generated and `null` is returned; otherwise, an `invalid` event is generated and an array of error strings is returned.

Example

The following example shows how to use `validateProperty()` to make sure that text entered by a user is of a valid length. You'll determine the valid length by setting the Validation Options for the String data type in the Component inspector's Schema tab. If the user enters a string of invalid length in the text field, the error messages returned by `validateProperty()` are displayed in the Output panel.

To validate text entered by a user in a TextInput component:

1. Drag a TextInput component from the Components panel to the Stage, and name it **zipCode_txt**.
2. Select the TextInput component and, in the Component inspector, click the Schema tab.
3. In the Schema Tree pane (the top pane of the Schema tab), select the `text` property.
4. In the Schema Attributes pane (the bottom pane of the Schema tab), select ZipCode from the Data Type pop-up menu.
5. Open the Timeline if it is not already open.
6. Click the first frame on Layer 1 in the Timeline, and open the Actions panel (Window > Development Panels > Actions).

7. Add the following code to the Actions panel:

```
// Add ComponentMixin methods to TextInput component.
// Note that this step is only necessary if the component
// isn't already involved in a data binding,
// either as the source or destination.
mx.data.binding.ComponentMixins.initComponent(zipCode_txt);
// Define event listener function for component:
validateResults = function (eventObj) {
  var errors:Array = eventObj.target.validateProperty("text");
  if (errors != null) {
    trace(errors);
  }
};
// Register listener function with component:
zipCode_txt.addEventListener("enter", validateResults);
```

8. Select Window > Other Panels > Common Libraries > Classes to open the Classes library.
9. Open your document's library by choosing Window > Library.
10. Drag DataBindingClasses from the Classes library to your document's library.

    This step makes the data binding runtime classes available to the SWF file at runtime.
11. Test the SWF file by selecting Control > Test Movie.

    In the TextInput component on the Stage, enter an invalid United States zip code—for example, one that contains all letters, or one that contains fewer than five numbers. Notice the error messages displayed in the Output panel.

DataType class (Flash Professional only)

ActionScript Class Name mx.data.binding.DataType

The DataType class provides read and write access to data fields of a component property. To get a DataType object, you call the `ComponentMixins.getField()` method on a component. You can then call methods of the DataType object to get and set the value of the field.

If you get and set field values directly on the component instance instead of using DataType class methods, the data is provided in its "raw" form. In contrast, when you get or set field values using DataType methods, the values are processed according to the field's schema settings.

For example, the following code gets the value of a component's property directly and assigns it to a variable. The variable, `propVar`, contains whatever "raw" value is the current value of the property `propName`.

```
var propVar = myComponent.propName;
```

The next example gets the value of the same property by using the `DataType.getAsString()` method. In this case, the value assigned to `stringVar` is the value of `propName` after being processed according to its schema settings, and then returned as a string.

```
var dataTypeObj:mx.data.binding.DataType = myComponent.getField("propName");
var stringVar: String = dataTypeObj.getAsString();
```

For more information about how to specify a field's schema settings, see "Working with schemas in the Schema tab (Flash Professional only)" in *Using Flash*.

You can also use the methods of the DataType class to get or set fields in various data types. The DataType class automatically converts the raw data to the requested type, if possible. For example, in the code example above, the data that's retrieved is converted to the String type, even if the raw data is a different type.

The `ComponentMixins.getField()` method is available for components that have been included in a data binding (either as a source, destination, or an index), or that have been initialized with `ComponentMixins.initComponent()`. For more information, see "ComponentMixins class (Flash Professional only)" on page 229.

Note: To make this class available at runtime, you must include the data binding classes in your FLA document.

For an overview of the classes in the mx.data.binding package, see "Classes in the mx.data.binding package (Flash Professional only)" on page 213.

Method summary for the DataType class

The following table lists the methods of the DataType class.

Method	Description
`DataType.getAnyTypedValue()`	Fetches the current value of the field.
`DataType.getAsBoolean()`	Fetches the current value of the field as a Boolean value.
`DataType.getAsNumber()`	Fetches the current value of the field as a number.
`DataType.getAsString()`	Fetches the current value of the field as a String value.
`DataType.getTypedValue()`	Fetches the current value of the field in the form of the requested data type.
`DataType.setAnyTypedValue()`	Sets a new value in the field.
`DataType.setAsBoolean()`	Sets the field to the new value, which is given as a Boolean value.
`DataType.setAsNumber()`	Sets the field to the new value, which is given as a number.
`DataType.setAsString()`	Sets the field to the new value, which is given as a string.
`DataType.setTypedValue()`	Sets a new value in the field.

Property summary for the DataType class

The following table lists the properties of the DataType class.

Property	Description
`DataType.encoder`	Provides a reference to the encoder object associated with this field.
`DataType.formatter`	Provides a reference to the formatter object associated with this field.
`DataType.kind`	Provides a reference to the Kind object associated with this field.

DataType.encoder

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.encoder

Description

Property; provides a reference to the encoder object associated with this field, if one exists. You can use this property to access any properties and methods defined by the specific encoder applied to the field in the Component inspector's Schema tab.

If no encoder was applied to the field in question, then this property returns `undefined`.

For more information about the encoders provided with Flash MX Professional 2004, see "Schema encoders" in *Using Flash*.

Example

The following example assumes that the field being accessed (`isValid`) uses the Boolean encoder (`mx.data.encoders.Bool`). This encoder is provided with Flash MX Professional 2004 and contains a property named `trueStrings` that specifies which strings should be interpreted as `true` values. The code below sets the `trueStrings` property for a field's encoder to be the strings "Yes" and "Oui".

```
var myField:mx.data.binding.DataType = dataSet.getField("isValid");
myField.encoder.trueStrings = "Yes,Oui";
```

DataType.formatter

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.formatter

Description

Property; provides a reference to the formatter object associated with this field, if one exists. You can use this property to access any properties and methods for the formatter object applied to the field in the Component inspector's Schema tab.

If no formatter was applied to the field in question, this property returns `undefined`.

For more information about the formatters provided with Flash MX Professional 2004, see "Schema formatters" in *Using Flash*.

Example

This example assumes that the field being accessed is using the Number Formatter (mx.data.formatters.NumberFormatter) provided with Flash MX Professional 2004. This formatter contains a property named `precision` that specifies how many digits to display after the decimal point. This code sets the `precision` property to two decimal places for a field using this formatter.

```
var myField:DataType = dataGrid.getField("currentBalance");
myField.formatter.precision = 2;
```

DataType.getAnyTypedValue()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`dataTypeObject``.getAnyTypedValue(`*`suggestedTypes`*`)`

Parameters

`suggestedTypes` An array of strings that specify, in descending order of desirability, the preferred data types for the field.

Returns

The current value of the field, in the form of one of the data types specified in the *`suggestedTypes`* array.

Description

Method; fetches the current value of the field, using the information in the field's schema to process the value. If the field can provide a value as the first data type specified in the *`suggestedTypes`* array, the method returns the field's value as that data type. If not, the method attempts to extract the field's value as the second data type specified in the *`suggestedTypes`* array, and so on.

If you specify `null` as one of the items in the *`suggestedTypes`* array, the method returns the value of the field in the data type specified in the Schema tab of the Component inspector. Specifying `null` always results in a value being returned, so only use `null` at the end of the array.

If a value can't be returned in the form of the one of the suggested types, it is returned in the type specified in the Schema tab.

Example

This example attempts to get the value of a field (`productInfo.available`) in an XMLConnector component's `results` property first as a number or, if that fails, as a string.

```
import mx.data.binding.DataType;
import mx.data.binding.TypedValue;
var f: DataType = myXmlConnector.getField("results", "productInfo.available");
var b: TypedValue = f.getAnyTypedValue(["Number", "String"]);
```

See also

`ComponentMixins.getField()`

DataType.getAsBoolean()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`dataTypeObject``.getAsBoolean()`

Parameters

None.

Returns

A Boolean value.

Description

Method; fetches the current value of the field and converts it to Boolean form, if necessary.

Example

In this example, a field named `propName` that belongs to a component named `myComponent` is retrieved as a Boolean value, and assigned to a variable:

```
var dataTypeObj:mx.data.binding.DataType = myComponent.getField("propName");
var propValue:Boolean = dataTypeObj.getAsBoolean();
```

DataType.getAsNumber()

Availability

Flash Player 6.

Edition

Flash MX Professional 2004.

Usage

`dataTypeObject``.getAsNumber()`

Parameters

None.

Returns

A number.

Description

Method; fetches the current value of the field and converts it to Number form, if necessary.

Example

In this example, a field named `propName` that belongs to a component named `myComponent` is retrieved as a number, and assigned to a variable:

```
var dataTypeObj:mx.data.binding.DataType = myComponent.getField("propName");
var propValue:Number = dataTypeObj.getAsNumber();
```

See also

`DataType.getAnyTypedValue()`

DataType.getAsString()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`dataTypeObject``.getAsString()`

Parameters

None.

Returns

A string.

Description

Method; fetches the current value of the field and converts it to String form, if necessary.

Example

In this example, a property named `propName` that belongs to a component named `myComponent` is retrieved as a string and assigned to a variable:

```
var dataTypeObj:mx.data.binding.DataType = myComponent.getField("propName");
var propValue:String = dataTypeObj.getAsString();
```

See also

`DataType.getAnyTypedValue()`

DataType.getTypedValue()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.getTypedValue(*requestedType*)

Parameters

requestedType A string containing the name of a data type, or `null`.

Returns

A TypedValue object (see "TypedValue class (Flash Professional only)" on page 244).

Description

Method; returns the value of the field in the specified form, if the field can provide its value in that form. If the field cannot provide its value in the requested form, the method returns `null`.

If `null` is specified as *requestedType*, the method returns the value of the field in its default type.

Example

The following example returns the value of the field converted to the Boolean data type. This is stored in the `bool` variable.

```
var bool:TypedValue = field.getTypedValue("Boolean");
```

DataType.kind

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.kind

Description

Property; provides a reference to the Kind object associated with this field. You can use this property to access properties and methods of the Kind object.

DataType.setAnyTypedValue()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
dataTypeObject.setAnyTypedValue(newTypedValue)
```

Parameters

newTypedValue A TypedValue object value to set in the field. For more information, see "TypedValue class (Flash Professional only)" on page 244.

Returns

An array of strings describing any errors that occurred while attempting to set the new value. Errors can occur under any of the following conditions:

- The data provided cannot be converted to the data type of this field (for example, `"abc"` cannot be converted to Number).
- The data is an acceptable type but does not meet the validation criteria of the field.
- The field is read-only.

Note: The actual text of an error message varies depending on the data type, formatters, and encoders that are defined in the field's schema.

Description

Method; sets a new value in the field, using the information in the field's schema to process the field.

This method operates by first calling `DataType.setTypedValue()` to set the value. If that fails, the method checks to see if the destination object is willing to accept String, Boolean, or Number data, and if so, attempts to use the corresponding ActionScript conversion functions.

Example

This example creates a new TypedValue object (a Boolean), and then assigns that value to a DataType object named `field`. Any errors that occur are assigned to the `errors` array.

```
import mx.data.binding.*;
var t:TypedValue = new TypedValue (true, "Boolean");
var errors: Array = field.setAnyTypedValue (t);
```

See also

`DataType.setTypedValue()`

DataType.setAsBoolean()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.setAsBoolean(*newBooleanValue*)

Parameters

newBooleanValue A Boolean value.

Returns

Nothing.

Description

Method; sets the field to the new value, which is given as a Boolean value. The value is converted to, and stored as, the data type that is appropriate for this field.

Example

The following example sets a variable named `bool` to the Boolean value `true`. It then sets the value referenced by `field` to `true`.

```
var bool: Boolean = true;
field.setAsBoolean (bool);
```

DataType.setAsNumber()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.setAsNumber(*newNumberValue*)

Parameters

newNumberValue A number.

Returns

Nothing.

Description

Method; sets the field to the new value, which is given as a number. The value is converted to, and stored as, the data type that is appropriate for this field.

Example

The following example sets a variable named `num` to the Number value of 32. It then sets the value referenced by `field` to `num`.

```
var num: Number = 32;
field.setAsNumber (num);
```

DataType.setAsString()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`dataTypeObject``.setAsString(`*`newStringValue`*`)`

Parameters

`newStringValue` A string.

Returns

Nothing.

Description

Method; sets the field to the new value, which is given as a string. The value is converted to, and stored as, the data type that is appropriate for this field.

Example

The following example sets the variable `stringVal` to the string `"The new value"`. It then sets the value of `field` to the string.

```
var stringVal: String = "The new value";
field.setAsString (stringVal);
```

DataType.setTypedValue()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`dataTypeObject``.setTypedValue(`*`newTypedValue`*`)`

Parameters

`newTypedValue` A TypedValue object value to set in the field.

For more information about TypedValue objects, see "TypedValue class (Flash Professional only)" on page 244.

Returns

An array of strings describing any errors that occurred while attempting to set the new value. Errors can occur under any of the following conditions:

- The data provided is not an acceptable type.
- The data provided cannot be converted to the data type of this field (for example, `"abc"` cannot be converted to Number).
- The data is an acceptable type but does not meet the validation criteria of the field.
- The field is read-only.

Note: The actual text of an error message varies depending on the data type, formatters, and encoders that are defined in the field's schema.

Description

Method; sets a new value in the field, using the information in the field's schema to process the field. This method behaves similarly to `DataType.setAnyTypedValue()`, except that it doesn't try as hard to convert the data to an acceptable data type. For more information, see `DataType.setAnyTypedValue()`.

Example

This example creates a new TypedValue object (a Boolean), and then assigns that value to a DataType object named `field`. Any errors that occur are assigned to the `errors` array.

```
import mx.data.binding.*;
var bool:TypedValue = new TypedValue (true, "Boolean");
var errors: Array = field.setTypedValue (bool);
```

See also

`DataType.setTypedValue()`

TypedValue class (Flash Professional only)

ActionScript Class Name mx.data.binding.TypedValue

A TypedValue object contains a data value, along with information about the value's data type. TypedValue objects are provided as parameters to, and are returned from, various methods of the DataType class. The data type information in the TypedValue object helps DataType objects decide when and how they need to do type conversion.

Note: To make this class available at runtime, you must include the data binding classes in your FLA document.

For an overview of the classes in the mx.data.binding package, see "Classes in the mx.data.binding package (Flash Professional only)" on page 213.

Property summary for the TypedValue class

The following table lists the properties of the TypedValue class.

Property	Description
`TypedValue.type`	Contains the schema associated with the TypedValue object's value.
`TypedValue.typeName`	Names the data type of the TypedValue object's value.
`TypedValue.value`	Contains the data value of the TypedValue object.

Constructor for the TypedValue class

Availability

Flash Player 6 (6.0 79.0).

Usage

```
new mx.data.binding.TypedValue(value, typeName, [type])
```

Parameters

`value` A data value of any type.

`typeName` A string that contains the name of the value's data type.

`type` An optional Schema object that describes in more detail the schema of the data. This field is required only in certain circumstances, such as when setting data into a DataSet component's `dataProvider` property.

Description

Constructor; creates a new TypedValue object.

TypedValue.type

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
typedValueObject.type
```

Description

Property; contains the schema associated with the TypedValue object's value.

Example

This example displays `null` in the Output panel:

```
var t: TypedValue = new TypedValue (true, "Boolean", null);
trace(t.type);
```

TypedValue.typeName

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

typedValueObject.typeName

Description

Property; contains the name of the data type of the TypedValue object's value.

Example

This example displays `Boolean` in the Output panel:

```
var t: TypedValue = new TypedValue (true, "Boolean", null);
trace(t.typeName);
```

TypedValue.value

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

typedValueObject.value

Description

Property; contains the data value of the TypedValue object.

Example

This example displays `true` in the Output panel:

```
var t: TypedValue = new TypedValue (true, "Boolean", null);
trace(t.value);
```

DataGrid component (Flash Professional only)

The DataGrid component lets you create powerful data-enabled displays and applications. You can use the DataGrid component to instantiate a recordset (retrieved from a database query in Macromedia ColdFusion, Java, or .Net) using Macromedia Flash Remoting and display it in columns. You can also use data from a data set or from an array to fill a DataGrid component. The version 2 DataGrid component has been improved to include horizontal scrolling, better event support (including event support for editable cells), enhanced sorting capabilities, and performance optimizations.

You can resize and customize characteristics such as the font, color, and borders of columns in a grid. You can use a custom movie clip as a "cell renderer" for any column in a grid. (A cell renderer displays the contents of a cell.) You can use scroll bars to move through data in a grid; you can also turn off scroll bars and use the DataGrid methods to create a page view style display.

When you add the DataGrid component to an application, you can use the Accessibility panel to make the component accessible to screen readers. First, you must add the following line of code to enable accessibility for the DataGrid component:

```
mx.accessibility.DataGridAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Interacting with the DataGrid component (Flash Professional only)

You can use the mouse and the keyboard to interact with a DataGrid component.

If `DataGrid.sortableColumns` and `DataGridColumn.sortOnHeaderRelease` are both `true`, clicking in a column header causes the grid to sort based on the column's cell values.

If `DataGrid.resizableColumns` is `true`, clicking in the area between columns lets you resize columns.

Clicking in an editable cell sends focus to that cell; clicking a non-editable cell has no effect on focus. An individual cell is editable when both the `DataGrid.editable` and `DataGridColumn.editable` properties of the cell are `true`.

When a DataGrid instance has focus either from clicking or tabbing, you can use the following keys to control it:

Key	Description
Down Arrow	When a cell is being edited, the insertion point shifts to the end of the cell's text. If a cell is not editable, the Down Arrow key handles selection as the List component does.
Up Arrow	When a cell is being edited, the insertion point shifts to the beginning of the cell's text. If a cell is not editable, the Up Arrow key handles selection as the List component does.
Right Arrow	When a cell is being edited, the insertion point shifts one character to the right. If a cell is not editable, the Right Arrow key does nothing.

Key	Description
Left Arrow	When a cell is being edited, the insertion point shifts one character to the left. If a cell is not editable, the Left Arrow key does nothing.
Return/Enter/Shift+Enter	When a cell is editable, the change is committed, and the insertion point is moved to the cell on the same column, next row (up or down, depending on the shift toggle).
Shift+Tab/Tab	Moves focus to the previous item. When the Tab key is pressed, focus cycles from the last column in the grid to the first column on the next line. When Shift+Tab is pressed, cycling is reversed. All the text in the focused cell is selected.

Using the DataGrid component (Flash Professional only)

You can use the DataGrid component as the foundation for numerous types of data-driven applications. You can easily display a formatted tabular view of a database query (or other data), but you can also use the cell renderer capabilities to build more sophisticated and editable user interface pieces. The following are practical uses for the DataGrid component:

- A webmail client
- Search results pages
- Spreadsheet applications such as loan calculators and tax form applications

Understanding the design of the DataGrid component

The DataGrid component extends the List component. When you design an application with the DataGrid component, it is helpful to understand how the List class underlying it was designed. The following are some fundamental assumptions and requirements that Macromedia used when developing the List class:

- Keep it small, fast, and simple.

  Don't make something more complicated than absolutely necessary. This was the prime design directive. Most of the requirements listed below are based on this directive.
- Lists have uniform row heights.

  Every row must be the same height; the height can be set during authoring or at runtime.
- Lists must scale to thousands of records.
- Lists don't measure text.

  This creates a horizontal scrolling issue for List and Tree components; for more information, see "Understanding the design of the List component" on page 451. The DataGrid component, however, supports `"auto"` as an `hScrollPolicy` value, because it measures columns (which are the same width per item), not text.

  The fact that lists don't measure text explains why lists have uniform row heights. Sizing individual rows to fit text would require intensive measuring. For example, if you wanted to accurately show the scroll bars on a list with nonuniform row height, you'd need to premeasure every row.

- Lists perform worse as a function of their visible rows.

  Although lists can display 5000 records, they can't render 5000 records at once. The more visible rows (specified by the `rowCount` property) you have on the Stage, the more work the list must to do to render. Limiting the number of visible rows, if at all possible, is the best solution.

- Lists aren't tables.

  DataGrid components are intended to provide an interface for many records. They're not designed to display complete information; they're designed to display enough information so that users can drill down to see more. The message view in Microsoft Outlook is a prime example. You don't read the entire e-mail in the grid; the message would be difficult to read and the client would perform terribly. Outlook displays enough information so that a user can drill into the post to see the details.

Understanding the DataGrid component: data model and view

Conceptually, the DataGrid component is composed of a data model and a view that displays the data. The data model consists of three main parts:

- DataProvider

  This is a list of items with which to fill the data grid. Any array in the same frame as a DataGrid component is automatically given methods (from the DataProvider API) that let you manipulate data and broadcast changes to multiple views. Any object that implements the DataProvider API can be assigned to the `DataGrid.dataProvider` property (including recordsets, data sets, and so on). The following code creates a data provider called `myDP`:

  ```
  myDP = new Array({name:"Chris", price:"Priceless"}, {name:"Nigel",
    price:"Cheap"});
  ```

- Item

  This is an ActionScript object used for storing the units of information in the cells of a column. A data grid is really a list that can display more than one column of data. A list can be thought of as an array; each indexed space of the list is an item. For the DataGrid component, each item consists of fields. In the following code, the content between curly braces (`{}`) is an item:

  ```
  myDP = new Array({name:"Chris", price:"Priceless"}, {name:"Nigel",
    price:"Cheap"});
  ```

- Field

  Identifiers that indicate the names of the columns within the items. This corresponds to the `columnNames` property in the columns list. In the List component, the fields are usually `label` and `data`, but in the DataGrid component the fields can be any identifier. In the following code, the fields are `name` and `price`:

  ```
  myDP = new Array({name:"Chris", price:"Priceless"}, {name:"Nigel",
    price:"Cheap"});
  ```

The view consists of three main parts:

- Row

  This is a view object responsible for rendering the items of the grid by laying out cells. Each row is laid out horizontally below the previous one.

- Column

  Columns are fields that are displayed in the grid; the fields each correspond to the `columnName` property of each column.

  Each column is a view object (an instance of the DataGridColumn class) responsible for displaying each column—for example, width, color, size, and so on.

  There are three ways to add columns to a data grid: assign a DataProvider object to `DataGrid.dataProvider` (this automatically generates a column for each field in the first item), set `DataGrid.columnNames` to specify which fields will be displayed, or use the constructor for the DataGridColumn class to create columns and call `DataGrid.addColumn()` to add them to the grid.

  To format columns, either set up style properties for the entire data grid, or define DataGridColumn objects, set up their style formats individually, and add them to the data grid.

- Cell

  This is a view object responsible for rendering the individual fields of each item. To communicate with the data grid, these components must implement the CellRenderer API (see "CellRenderer API" on page 145). For a basic data grid, a cell is a built-in ActionScript TextField object.

DataGrid parameters

You can set the following authoring parameters for each DataGrid component instance in the Property inspector or in the Component inspector:

multipleSelection is a Boolean value that indicates whether multiple items can be selected (`true`) or not (`false`). The default value is `false`.

rowHeight indicates the height of each row, in pixels. Changing the font size does not change the row height. The default value is 20.

editable is a Boolean value that indicates whether the grid is editable (`true`) or not (`false`). The default value is `false`.

You can write ActionScript to control these and additional options for the DataGrid component using its properties, methods, and events. For more information, see "DataGrid class (Flash Professional only)" on page 254.

Creating an application with the DataGrid component

To create an application with the DataGrid component, you must first determine where your data is coming from. The data for a grid can come from a recordset that is fed from a database query in Macromedia ColdFusion, Java, or .Net using Flash Remoting. Data can also come from a data set or an array. To pull the data into a grid, you set the `DataGrid.dataProvider` property to the recordset, data set, or array. You can also use the methods of the DataGrid and DataGridColumn classes to create data locally. Any Array object in the same frame as a DataGrid component copies the methods, properties, and events of the DataProvider API.

To use Flash Remoting to add a DataGrid component to an application:

1. In Flash, select File > New and select Flash Document.
2. In the Components panel, double-click the DataGrid component to add it to the Stage.
3. In the Property inspector, enter the instance name **myDataGrid**.
4. In the Actions panel on Frame 1, enter the following code:

   ```
   myDataGrid.dataProvider = recordSetInstance;
   ```

   The Flash Remoting recordset `recordSetInstance` is assigned to the `dataProvider` property of `myDataGrid`.
5. Select Control > Test Movie.

To use a local data provider to add a DataGrid component to an application:

1. In Flash, select File > New and select Flash Document.
2. In the Components panel, double-click the DataGrid component to add it to the Stage.
3. In the Property inspector, enter the instance name **myDataGrid**.
4. In the Actions panel on Frame 1, enter the following code:

   ```
   myDP = new Array({name:"Chris", price:"Priceless"}, {name:"Nigel",
     price:"Cheap"});
   myDataGrid.dataProvider = myDP;
   ```

   The `name` and `price` fields are used as the column headings, and their values fill the cells in each row.
5. Select Control > Test Movie.

Customizing the DataGrid component (Flash Professional only)

You can transform a DataGrid component horizontally and vertically during authoring and runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`). If there is no horizontal scroll bar, column widths adjust proportionally. If column (and therefore, cell) size adjustment occurs, text in the cells may be clipped.

Using styles with the DataGrid component

You can set style properties to change the appearance of a DataGrid component. The DataGrid component inherits styles from the List component. (See "Using styles with the List component" on page 453.) The DataGrid component also supports the following styles:

Style	Theme	Description
`backgroundColor`	Both	The background color, which can be set for the whole grid or for each column.
`backgroundDisabledColor`	Both	The background color when the component's `enabled` property is set to `"false"`. The default value is 0xDDDDDD (medium gray).
`border styles`	Both	The DataGrid component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style value is `"inset"`.
`headerColor`	Both	The color of the column headers. The default value is 0xEAEAEA (light gray)
`headerStyle`	Both	A CSS style declaration for the column header that can be applied to a grid or column to customize the header styles.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textAlign`	Both	The text alignment: either `"left"`, `"right"`, or `"center"`. The default value is `"left"`.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.
`vGridLines`	Both	A Boolean value that indicates whether to show vertical grid lines (`true`) or not (`false`). The default value is `true`.

Style	Theme	Description
hGridLines	Both	A Boolean value that indicates whether to show horizontal grid lines (true) or not (false). The default value is false.
vGridLineColor	Both	The color of the vertical grid lines. The default value is 0x666666 (medium gray).
hGridLineColor	Both	The color of the horizontal grid lines. The default value is 0x666666 (medium gray).

Setting styles for an individual column

Color and text styles can be set for the grid as a whole or for a column. You can use the following syntax to set a style for a particular column:

```
grid.getColumnAt(3).setStyle("backgroundColor", 0xFF00AA);
```

Setting header styles

You can set header styles through headerStyle, which is a style property itself. To do this, you create an instance of CSSStyleDeclaration, set the appropriate properties on that instance for the header, and then assign the CSSStyleDeclaration to the headerStyle property, as shown in the following example.

```
import mx.styles.CSSStyleDeclaration;
var headerStyles = new CSSStyleDeclaration();
headerStyles.setStyle("fontStyle", "italic");
grid.setStyle("headerStyle", headerStyles);
```

Setting styles for all DataGrid components in a document

The DataGrid class inherits from the List class, which inherits from the ScrollSelectList class. The default class-level style properties are defined on the ScrollSelectList class, which the Menu component and all List-based components extend. You can set new default style values on this class directly, and these new settings will be reflected in all affected components.

```
_global.styles.ScrollSelectList.setStyle("backgroundColor", 0xFF00AA);
```

To set a style property on the DataGrid components only, you can create a new instance of CSSStyleDeclaration and store it in _global.styles.DataGrid.

```
import mx.styles.CSSStyleDeclaration;
if (_global.styles.DataGrid == undefined) {
  _global.styles.DataGrid = new CSSStyleDeclaration();
}
_global.styles.DataGrid.setStyle("backgroundColor", 0xFF00AA);
```

When creating a new class-level style declaration, you will lose all default values provided by the ScrollSelectList declaration, including backgroundColor, which is required for supporting mouse events. To create a class-level style declaration and preserve defaults, use a for..in loop to copy the old settings to the new declaration.

```
var source = _global.styles.ScrollSelectList;
var target = _global.styles.DataGrid;
for (var style in source) {
```

```
  target.setStyle(style, source.getStyle(style));
}
```

For more information about class-level styles, see "Setting styles for a component class" on page 71.

Using skins with the DataGrid component

The skins that the DataGrid component uses to represent its visual states are included in the subcomponents that constitute the data grid (scroll bars and RectBorder). For information about their skins, see "Using skins with the UIScrollBar component" on page 831 and "RectBorder class" on page 647.

DataGrid class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > ScrollSelectList > List component > DataGrid

ActionScript Class Name mx.controls.DataGrid

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.DataGrid.version);
```

Note: The code `trace(myDataGridInstance.version);` returns `undefined`.

Method summary for the DataGrid class

The following table lists methods of the DataGrid class.

Method	Description
`DataGrid.addColumn()`	Adds a column to the data grid.
`DataGrid.addColumnAt()`	Adds a column to the data grid at a specified location.
`DataGrid.addItem()`	Adds an item to the data grid.
`DataGrid.addItemAt()`	Adds an item to the data grid at a specified location.
`DataGrid.editField()`	Replaces the cell data at a specified location.
`DataGrid.getColumnAt()`	Gets a reference to a column at a specified location.
`DataGrid.getColumnIndex()`	Gets a reference to the DataGridColumn object at the specified index.
`DataGrid.removeAllColumns()`	Removes all columns from a data grid.
`DataGrid.removeColumnAt()`	Removes a column from a data grid at a specified location.
`DataGrid.replaceItemAt()`	Replaces an item at a specified location with another item.
`DataGrid.spaceColumnsEqually()`	Spaces all columns equally.

Methods inherited from the UIObject class

The following table lists the methods the DataGrid class inherits from the UIObject class. When calling these methods, use the form *dataGridInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the DataGrid class inherits from the UIComponent class. When calling these methods, use the form *dataGridInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Methods inherited from the List class

The following table lists the methods the DataGrid class inherits from the List class. When calling these methods, use the form *dataGridInstance.methodName*.

Method	Description
List.addItem()	Adds an item to the end of the list.
List.addItemAt()	Adds an item to the list at the specified index.
List.getItemAt()	Returns the item at the specified index.
List.removeAll()	Removes all items from the list.
List.removeItemAt()	Removes the item at the specified index.
List.replaceItemAt()	Replaces the item at the specified index with another item.
List.setPropertiesAt()	Applies the specified properties to the specified item.

Method	Description
List.sortItems()	Sorts the items in the list according to the specified compare function.
List.sortItemsBy()	Sorts the items in the list according to a specified property.

Property summary for the DataGrid class

The following table lists the properties of the DataGrid class.

Property	Description
DataGrid.columnCount	Read-only; the number of columns that are displayed.
DataGrid.columnNames	An array of field names within each item that are displayed as columns.
DataGrid.dataProvider	The data model for a data grid.
DataGrid.editable	A Boolean value that indicates whether the data grid is editable (true) or not (false).
DataGrid.focusedCell	Defines the cell that has focus.
DataGrid.headerHeight	The height of the column headers, in pixels.
DataGrid.hScrollPolicy	Indicates whether a horizontal scroll bar is present ("on"), not present ("off"), or appears when necessary ("auto").
DataGrid.resizableColumns	A Boolean value that indicates whether the columns are resizable (true) or not (false).
DataGrid.selectable	A Boolean value that indicates whether the data grid is selectable (true) or not (false).
DataGrid.showHeaders	A Boolean value that indicates whether the column headers are visible (true) or not (false).
DataGrid.sortableColumns	A Boolean value that indicates whether the columns are sortable (true) or not (false).

Properties inherited from the UIObject class

The following table lists the properties the DataGrid class inherits from the UIObject class. When accessing these properties from the DataGrid object, use the form *dataGridInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.

Property	Description
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the DataGrid class inherits from the UIComponent class. When accessing these properties from the DataGrid object, use the form *dataGridInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the List class

The following table lists the properties the DataGrid class inherits from the List class. When accessing these properties from the DataGrid object, use the form *dataGridInstance.propertyName*.

Property	Description
List.cellRenderer	Assigns the class or symbol to use to display each row of the list.
List.dataProvider	The source of the list items.
List.hPosition	The horizontal position of the list.
List.hScrollPolicy	Indicates whether the horizontal scroll bar is displayed (`"on"`) or not (`"off"`).
List.iconField	A field in each item to be used to specify icons.
List.iconFunction	A function that determines which icon to use.
List.labelField	Specifies a field of each item to be used as label text.
List.labelFunction	A function that determines which fields of each item to use for the label text.
List.length	The number of items in the list. This property is read-only.

Property	Description
List.maxHPosition	The number of pixels the list can scroll to the right, when List.hScrollPolicy is set to "on".
List.multipleSelection	Indicates whether multiple selection is allowed in the list (true) or not (false).
List.rowCount	The number of rows that are at least partially visible in the list.
List.rowHeight	The pixel height of every row in the list.
List.selectable	Indicates whether the list is selectable (true) or not (false).
List.selectedIndex	The index of a selection in a single-selection list.
List.selectedIndices	An array of the selected items in a multiple-selection list.
List.selectedItem	The selected item in a single-selection list. This property is read-only.
List.selectedItems	The selected item objects in a multiple-selection list. This property is read-only.
List.vPosition	Scrolls the list so the topmost visible item is the number assigned.
List.vScrollPolicy	Indicates whether the vertical scroll bar is displayed ("on"), not displayed ("off"), or displayed when needed ("auto").

Event summary for the DataGrid class

The following table lists the events of the DataGrid class.

Event	Description
DataGrid.cellEdit	Broadcast when the cell value has changed.
DataGrid.cellFocusIn	Broadcast when a cell receives focus.
DataGrid.cellFocusOut	Broadcast when a cell loses focus.
DataGrid.cellPress	Broadcast when a cell is pressed (clicked).
DataGrid.change	Broadcast when an item has been selected.
DataGrid.columnStretch	Broadcast when a user resizes a column horizontally.
DataGrid.headerRelease	Broadcast when a user clicks (releases) a header.

Events inherited from the UIObject class

The following table lists the events the DataGrid class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.

Event	Description
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the DataGrid class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Events inherited from the List class

The following table lists the events the DataGrid class inherits from the List class.

Event	Description
List.change	Broadcast whenever user interaction causes the selection to change.
List.itemRollOut	Broadcast when the pointer rolls over and then off of list items.
List.itemRollOver	Broadcast when the pointer rolls over list items.
List.scroll	Broadcast when a list is scrolled.

DataGrid.addColumn()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.addColumn(*dataGridColumn*)

myDataGrid.addColumn(*name*)

Parameters

dataGridColumn An instance of the DataGridColumn class.

name A string that indicates the name of a new DataGridColumn object to be inserted.

Returns

A reference to the DataGridColumn object that was added.

Description

Method; adds a new column to the end of the data grid. For more information, see "DataGridColumn class (Flash Professional only)" on page 278.

Example

The following code adds a new DataGridColumn object named `Purple`:

```
import mx.controls.gridclasses.DataGridColumn;
myGrid.addColumn(new DataGridColumn("Purple"));
```

DataGrid.addColumnAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.addColumnAt(*index*, *name*)

myDataGrid.addColumnAt(*index*, *dataGridColumn*)

Parameters

index The index position at which the DataGridColumn object is added. The first position is 0.

name A string that indicates the name of the DataGridColumn object.

dataGridColumn An instance of the DataGridColumn class.

Returns

A reference to the DataGridColumn object that was added.

Description

Method; adds a new column at the specified position. Columns are shifted to the right and their indexes are incremented. For more information, see "DataGridColumn class (Flash Professional only)" on page 278.

Example

The following example inserts a new DataGridColumn object called `"Green"` at the second and fourth columns:

```
import mx.controls.gridclasses.DataGridColumn;
myGrid.addColumnAt(1, "Green");
myGrid.addColumnAt(3, new DataGridColumn("Purple"));
```

DataGrid.addItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.addItem(*item*)

Parameters

item An instance of an object to be added to the grid.

Returns

A reference to the instance that was added.

Description

Method; adds an item to the end of the grid (after the last item index).

Note: This differs from the `List.addItem()` method in that an object is passed rather than a string.

Example

The following example adds a new object to the grid `myGrid`:

```
var anObject= {name:"Jim!!", age:30};
myGrid.addItem(anObject);
```

DataGrid.addItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.addItemAt(*index*, *item*)

Parameters

index The index position (among the child nodes) at which the node should be added. The first position is 0.

item A string that displays the node.

Returns

A reference to the object instance that was added.

Description

Method; adds an item to the grid at the position specified.

Example

The following example inserts an object instance to the grid at index position 4:

```
var anObject= {name:"Jim!!", age:30};
myGrid.addItemAt(4, anObject);
```

DataGrid.cellEdit

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.cellEdit = function(eventObject){
  // insert your code here
}
myDataGridInstance.addEventListener("cellEdit", listenerObject)
```

Description

Event; broadcast to all registered listeners when cell value changes.

Version 2 components use a dispatcher/listener event model. The DataGrid component dispatches a `cellEdit` event when the value of a cell has changed, and the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `DataGrid.cellEdit` event's event object has four additional properties:

`columnIndex` A number that indicates the index of the target column.

`itemIndex` A number that indicates the index of the target row.

`oldValue` The previous value of the cell.

`type` The string `"cellEdit"`.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myDataGridListener` is defined and passed to `myDataGrid.addEventListener()` as the second parameter. The event object is captured by the `cellEdit` handler in the *`eventObject`* parameter. When the `cellEdit` event is broadcast, a `trace` statement is sent to the Output panel.

```
myDataGridListener = new Object();
myDataGridListener.cellEdit = function(event){
```

```
  var cell = "(" + event.columnIndex + ", " + event.itemIndex + ")";
  trace("The value of the cell at " + cell + " has changed");
}
myDataGrid.addEventListener("cellEdit", myDataGridListener);
```

Note: The grid must be editable for the above code to work.

DataGrid.cellFocusIn

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.cellFocusIn = function(eventObject){
  // insert your code here
}
myDataGridInstance.addEventListener("cellFocusIn", listenerObject)
```

Description

Event; broadcast to all registered listeners when a particular cell receives focus. This event is broadcast after any previously edited cell's `editCell` and `cellFocusOut` events are broadcast.

Version 2 components use a dispatcher/listener event model. When a DataGrid component dispatches a `cellFocusIn` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `DataGrid.cellFocusIn` event's event object has three additional properties:

`columnIndex` A number that indicates the index of the target column.

`itemIndex` A number that indicates the index of the target row.

`type` The string `"cellFocusIn"`.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myListener` is defined and passed to `grid.addEventListener()` as the second parameter. The event object is captured by the `cellFocusIn` handler in the *`eventObject`* parameter. When the `cellFocusIn` event is broadcast, a `trace` statement is sent to the Output panel.

```
var myListener = new Object();
myListener.cellFocusIn = function(event) {
    var cell = "(" + event.columnIndex + ", " + event.itemIndex + ")";
```

```
    trace("The cell at " + cell + " has gained focus");
};
grid.addEventListener("cellFocusIn", myListener);
```

Note: The grid must be editable for the above code to work.

DataGrid.cellFocusOut

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.cellFocusOut = function(eventObject){
  // insert your code here
}
myDataGridInstance.addEventListener("cellFocusOut", listenerObject)
```

Description

Event; broadcast to all registered listeners whenever a user moves off a cell that has focus. You can use the event object properties to isolate the cell that was left. This event is broadcast after the `cellEdit` event and before any subsequent `cellFocusIn` events are broadcast by the next cell.

Version 2 components use a dispatcher/listener event model. When a DataGrid component dispatches a `cellFocusOut` event, the event is handled by a function (also called a *handler*) that is attached to a listener object that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `DataGrid.cellFocusOut` event's event object has three additional properties:

`columnIndex` A number that indicates the index of the target column. The first position is 0.

`itemIndex` A number that indicates the index of the target row. The first position is 0.

`type` The string `"cellFocusOut"`.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myListener` is defined and passed to `grid.addEventListener()` as the second parameter. The event object is captured by the `cellFocusOut` handler in the *`eventObject`* parameter. When the `cellFocusOut` event is broadcast, a `trace` statement is sent to the Output panel.

```
var myListener = new Object();
myListener.cellFocusOut = function(event) {
    var cell = "(" + event.columnIndex + ", " + event.itemIndex + ")";
```

```
    trace("The cell at " + cell + " has lost focus");
};
grid.addEventListener("cellFocusOut", myListener);
```

Note: The grid must be editable for the above code to work.

DataGrid.cellPress

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.cellPress = function(eventObject){
  // insert your code here
}
myDataGridInstance.addEventListener("cellPress", listenerObject)
```

Description

Event; broadcast to all registered listeners when a user presses the mouse button on a cell.

Version 2 components use a dispatcher/listener event model. When a DataGrid component broadcasts a `cellPress` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `DataGrid.cellPress` event's event object has three additional properties:

`columnIndex` A number that indicates the index of the column that was pressed. The first position is 0.

`itemIndex` A number that indicates the index of the row that was pressed. The first position is 0.

`type` The string `"cellPress"`.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myListener` is defined and passed to `grid.addEventListener()` as the second parameter. The event object is captured by the `cellPress` handler in the *`eventObject`* parameter. When the `cellPress` event is broadcast, a `trace` statement is sent to the Output panel.

```
var myListener = new Object();
myListener.cellPress = function(event) {
    var cell = "(" + event.columnIndex + ", " + event.itemIndex + ")";
```

```
    trace("The cell at " + cell + " has been clicked");
};
grid.addEventListener("cellPress", myListener);
```

DataGrid.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  // insert your code here
}
myDataGridInstance.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners when an item has been selected.

Version 2 components use a dispatcher/listener event model. When a DataGrid component dispatches a `change` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `DataGrid.change` event's event object has one additional property, `type`, whose value is `"change"`. For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myListener` is defined and passed to `grid.addEventListener()` as the second parameter. The event object is captured by `change` handler in the *`eventObject`* parameter. When the `change` event is broadcast, a `trace` statement is sent to the Output panel.

```
var myListener = new Object();
myListener.change = function(event) {
    trace("The selection has changed to " + event.target.selectedIndex);
};
grid.addEventListener("change", myListener);
```

DataGrid.columnCount

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.columnCount
```

Description

Property (read-only); the number of columns displayed.

Example

The following example gets the number of displayed columns in the DataGrid instance `grid`:

```
var c = grid.columnCount;
```

DataGrid.columnNames

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.columnNames
```

Description

Property; an array of field names within each item that are displayed as columns.

Example

The following example tells the `grid` instance to display only these three fields as columns:

```
grid.columnNames = ["Name", "Description", "Price"];
```

DataGrid.columnStretch

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.columnStretch = function(eventObject){
  // insert your code here
}
myDataGridInstance.addEventListener("columnStretch", listenerObject)
```

Description

Event; broadcast to all registered listeners when a user resizes a column horizontally.

Version 2 components use a dispatcher/listener event model. When a DataGrid component dispatches a `columnStretch` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `DataGrid.columnStretch` event's event object has two additional properties:

`columnIndex` A number that indicates the index of the target column. The first position is 0.

`type` The string `"columnStretch"`.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myListener` is defined and passed to `grid.addEventListener()` as the second parameter. The event object is captured by the `columnStretch` handler in the *`eventObject`* parameter. When the `columnStretch` event is broadcast, a `trace` statement is sent to the Output panel.

```
var myListener = new Object();
myListener.columnStretch = function(event) {
    trace("column " + event.columnIndex + " was resized");
};
grid.addEventListener("columnStretch", myListener);
```

DataGrid.dataProvider

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.dataProvider
```

Description

Property; the data model for items viewed in a DataGrid component.

The data grid adds methods to the prototype of the Array class so that each Array object conforms to the DataProvider API (see DataProvider.as in the Classes/mx/controls/listclasses folder). Any array that is in the same frame or screen as a data grid automatically has all the methods (`addItem()`, `getItemAt()`, and so on) needed for it to be the data model of a data grid, and can be used to broadcast data model changes to multiple components.

In a DataGrid component, you specify fields for display in the `DataGrid.columnNames` property. If you don't define the column set (by setting the `DataGrid.columnNames` property or by calling `DataGrid.addColumn()`) for the data grid before the `DataGrid.dataProvider` property has been set, the data grid generates columns for each field in the data provider's first item, once that item arrives.

Any object that implements the DataProvider API can be used as a data provider for a data grid (including Flash Remoting recordsets, data sets, and arrays).

Use a grid's data provider to communicate with the data in the grid because the data provider remains consistent, regardless of scroll position.

Example

The following example creates an array to be used as a data provider and assigns it directly to the `dataProvider` property:

```
grid.dataProvider = [{name:"Chris", price:"Priceless"}, {name:"Nigel",
  price:"cheap"}];
```

The following example creates a new Array object that is decorated with the DataProvider API. It uses a `for` loop to add 20 items to the grid:

```
myDP = new Array();
for (var i=0; i<20; i++)
  myDP.addItem({name:"Nivesh", price:"Priceless"});
list.dataProvider = myDP
```

DataGrid.editable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid`.editable`

Description

Property; determines whether the data grid can be edited by a user (`true`) or not (`false`). This property must be `true` in order for individual columns to be editable and for any cell to receive focus. The default value is `false`.

If you want individual columns to be uneditable, use the `DataGridColumn.editable` property.

Caution: The DataGrid is not editable or sortable if it is bound directly to a WebServiceConnector component or an XMLConnector component. You must bind the DataGrid component to the DataSet component and bind the DataSet component to the WebServiceConnector component or XMLConnector component if you want the grid to be editable or sortable. For more information, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Example

The following example allows users to edit all the columns of the grid except the first column:

```
myDataGrid.editable = true;
myDataGrid.getColumnAt(0).editable = false;
```

See also

`DataGridColumn.editable`

DataGrid.editField()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`myDataGrid``.editField(`*`index, colName, data`*`)`

Parameters

`index` The index of the target cell. This number is zero-based.

`colName` A string indicating the name of the column (field) that contains the target cell.

`data` The value to be stored in the target cell. This parameter can be of any data type.

Returns

The data that was in the cell.

Description

Method; replaces the cell data at the specified location and refreshes the data grid with the new value. Any cell present for that value will have its `setValue()` method triggered.

Example

The following example places a value in the grid:

```
var prevValue = myGrid.editField(5, "Name", "Neo");
```

DataGrid.focusedCell

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.focusedCell

Description

Property; in editable mode only, an object instance that defines the cell that has focus. The object must have the fields `columnIndex` and `itemIndex`, which are both integers that indicate the index of the column and item of the cell. The origin is (0,0). The default value is `undefined`.

Example

The following example sets the focused cell to the third column, fourth row:

```
grid.focusedCell = {columnIndex:2, itemIndex:3};
```

DataGrid.getColumnAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.getColumnAt(*index*)

Parameters

index The index of the DataGridColumn object to be returned. This number is zero-based.

Returns

A DataGridColumn object.

Description

Method; gets a reference to the DataGridColumn object at the specified index.

Example

The following example gets the DataGridColumn object at index 4:

```
var aColumn = myGrid.getColumnAt(4);
```

DataGrid.getColumnIndex()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnIndex(columnName)
```

Parameters

columnName A string that is the name of a column.

Returns

A number that specifies the index of the column.

Description

Method; returns the index of the column specified by the *columnName* parameter.

DataGrid.headerHeight

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.headerHeight
```

Description

Property; the height of the header bar of the data grid, in pixels. The default value is 20.

Example

The following example sets the scroll position to the top of the display:

```
myDataGrid.headerHeight = 30;
```

DataGrid.headerRelease

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.headerRelease = function(eventObject){
  // insert your code here
}
myDataGridInstance.addEventListener("headerRelease", listenerObject)
```

Description

Event; broadcast to all registered listeners when a column header has been released. You can use this event with the `DataGridColumn.sortOnHeaderRelease` property to prevent automatic sorting and to let you sort as you like.

Version 2 components use a dispatcher/listener event model. When the DataGrid component dispatches a `headerRelease` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `DataGrid.headerRelease` event's event object has two additional properties:

`columnIndex` A number that indicates the index of the target column.

`type` The string `"headerRelease"`.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myListener` is defined and passed to `grid.addEventListener()` as the second parameter. The event object is captured by the `headerRelease` handler in the *`eventObject`* parameter. When the `headerRelease` event is broadcast, a `trace` statement is sent to the Output panel.

```
var myListener = new Object();
myListener.headerRelease = function(event) {
    trace("column " + event.columnIndex + " header was pressed");
};
grid.addEventListener("headerRelease", myListener);
```

DataGrid.hScrollPolicy

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.hScrollPolicy
```

Description

Property; specifies whether the data grid has a horizontal scroll bar. This property can have the value `"on"`, `"off"`, or `"auto"`. The default value is `"off"`.

If `hScrollPolicy` is set to `"off"`, columns scale proportionally to accommodate the finite width.

Note: This differs from the List component, which cannot have `hScrollPolicy` set to `"auto"`.

Example

The following example sets horizontal scroll policy to automatic, which means that the horizontal scroll bar appears if it's necessary to display all the content:

```
myDataGrid.hScrollPolicy = "auto";
```

DataGrid.removeAllColumns()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid`.removeAllColumns()`

Parameters

None.

Returns

Nothing.

Description

Method; removes all DataGridColumn objects from the data grid. Calling this method has no effect on the data provider.

Call this method if you are setting a new data provider that has different fields from the previous data provider, and you want to clear the fields that are displayed.

Example

The following example removes all DataGridColumn objects from `myDataGrid`:

```
myDataGrid.removeAllColumns();
```

DataGrid.removeColumnAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.removeColumnAt(index)
```

Parameters

index The index of the column to remove.

Returns

A reference to the DataGridColumn object that was removed.

Description

Method; removes the DataGridColumn object at the specified index.

Example

The following example removes the DataGridColumn object at index 2 in `myDataGrid`:

```
myDataGrid.removeColumnAt(2);
```

DataGrid.replaceItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.replaceItemAt(index, item)
```

Parameters

index The index of the item to be replaced.

item An object that is the item value to use as a replacement.

Returns

The previous value.

Description

Method; replaces the item at a specified index and refreshes the display of the grid.

Example

The following example replaces the item at index 4 with the item defined in `aNewValue`:

```
var aNewValue = {name:"Jim", value:"tired"};
var prevValue = myGrid.replaceItemAt(4, aNewValue);
```

DataGrid.resizableColumns

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.resizableColumns

Description

Property; a Boolean value that determines whether the columns of the grid can be stretched by the user (`true`) or not (`false`). This property must be `true` for individual columns to be resizable by the user. The default value is `true`.

Example

The following example prevents users from resizing columns:

```
myDataGrid.resizableColumns = false;
```

DataGrid.selectable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.selectable

Description

Property; a Boolean value that determines whether a user can select the data grid (`true`) or not (`false`). The default value is `true`.

Example

The following example prevents the grid from being selected:

```
myDataGrid.selectable = false;
```

DataGrid.showHeaders

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.showHeaders

Description

Property; a Boolean value that indicates whether the data grid displays the column headers (`true`) or not (`false`). Column headers are shaded to differentiate them from the other rows in a grid. Users can click column headers to sort the contents of the column if `DataGrid.sortableColumns` is set to `true`. The default value of `showHeaders` is `true`.

Example

The following example hides the column headers:

```
myDataGrid.showHeaders = false;
```

See also

`DataGrid.sortableColumns`

DataGrid.sortableColumns

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.sortableColumns

Description

Property; a Boolean value that determines whether the columns of the data grid can be sorted (`true`) or not (`false`) when a user clicks the column headers. This property must be `true` for individual columns to be sortable, and for the `headerRelease` event to be broadcast. The default value is `true`.

Caution: The DataGrid is not editable or sortable if it is bound directly to a WebServiceConnector component or an XMLConnector component. You must bind the DataGrid component to the DataSet component and bind the DataSet component to the WebServiceConnector component or XMLConnector component if you want the grid to be editable or sortable. For more information, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Example

The following example turns off sorting:

```
myDataGrid.sortableColumns = false;
```

See also

`DataGrid.headerRelease`

DataGrid.spaceColumnsEqually()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.spaceColumnsEqually()

Parameters

None.

Returns

Nothing.

Description

Method; respaces the columns equally.

Example

The following example respaces the columns of myGrid when any column header is pressed and released:

```
myGrid.showHeaders = true
myGrid.dataProvider = [{guitar:"Flying V", name:"maggot"}, {guitar:"SG",
  name:"dreschie"}, {guitar:"jagstang", name:"vitapup"}];
gridLO = new Object();
gridLO.headerRelease = function(){
  myGrid.spaceColumnsEqually();
}
myGrid.addEventListener("headerRelease", gridLO);
```

DataGridColumn class (Flash Professional only)

ActionScript Class Name mx.controls.gridclasses.DataGridColumn

You can create and configure DataGridColumn objects to use as columns of a data grid. Many of the methods of the DataGrid class are dedicated to managing DataGridColumn objects. DataGridColumn objects are stored in an zero-based array in the data grid; 0 is the leftmost column. After columns have been added or created, you can access them by calling DataGrid.getColumnAt(*index*).

There are three ways to add or create columns in a grid. If you want to configure your columns, it is best to use either the second or third way before you add data to a data grid so you don't have to create columns twice.

- Adding a data provider or an item with multiple fields to a grid that has no configured DataGridColumn objects automatically generates columns for every field in the reverse order of the for..in loop.

- `DataGrid.columnNames` takes in the field names of the desired item fields and generates DataGridColumn objects, in order, for each field listed. This approach lets you select and order columns quickly with a minimal amount of configuration. This approach removes any previous column information.
- The most flexible way to add columns is to prebuild them as DataGridColumn objects and add them to the data grid by using `DataGrid.addColumn()`. This approach is useful because it lets you add columns with proper sizing and formatting before the columns ever reach the grid (which reduces processor demand). For more information, see "Constructor for the DataGridColumn class" on page 279.

Property summary for the DataGridColumn class

The following table lists the properties of the DataGridColumn class.

Property	Description
`DataGridColumn.cellRenderer`	The linkage identifier of a symbol to be used to display the cells in this column.
`DataGridColumn.columnName`	Read-only; the name of the field associated with the column.
`DataGridColumn.editable`	A Boolean value that indicates whether a column is editable (`true`) or not (`false`).
`DataGridColumn.headerRenderer`	The name of a class to be used to display the header of this column.
`DataGridColumn.headerText`	The text for the header of this column.
`DataGridColumn.labelFunction`	A function that determines which field of an item to display.
`DataGridColumn.resizable`	A Boolean value that indicates whether a column is resizable (`true`) or not (`false`).
`DataGridColumn.sortable`	A Boolean value that indicates whether a column is sortable (`true`) or not (`false`).
`DataGridColumn.sortOnHeaderRelease`	A Boolean value that indicates whether a column is sorted (`true`) or not (`false`) when a user clicks a column header.
`DataGridColumn.width`	The width of a column, in pixels.

Constructor for the DataGridColumn class

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
new DataGridColumn(name)
```

Parameters

name A string that indicates the name of the DataGridColumn object. This parameter is the field of each item to display.

Returns

Nothing.

Description

Constructor; creates a DataGridColumn object. Use this constructor to create columns to add to a DataGrid component. After you create the DataGridColumn objects, you can add them to a data grid by calling `DataGrid.addColumn()`.

Example

The following example creates a DataGridColumn object called `Location`:

```
import mx.controls.gridclasses.DataGridColumn;
var column = new DataGridColumn("Location");
```

DataGridColumn.cellRenderer

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.getColumnAt(*index*).cellRenderer

Description

Property; a linkage identifier for a symbol to be used to display cells in this column. Any class used for this property must implement the CellRenderer API (see "CellRenderer API" on page 145.) The default value is `undefined`.

Example

The following example uses a linkage identifier to set a new cell renderer:

```
myGrid.getColumnAt(3).cellRenderer = "MyCellRenderer";
```

DataGridColumn.columnName

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.getColumnAt(*index*).columnName

Description

Property (read-only); the name of the field associated with this column. The default value is the name called in the DataGridColumn constructor.

Example

The following example assigns the column name of the column at the third index position to the variable `name`:

```
var name = myGrid.getColumnAt(3).columnName;
```

See also

Constructor for the DataGridColumn class

DataGridColumn.editable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDataGrid.getColumnAt(*index*).editable

Description

Property; determines whether the column can be edited by a user (`true`) or not (`false`). The `DataGrid.editable` property must be `true` in order for individual columns to be editable, even when `DataGridColumn.editable` is set to `true`. The default value is `true`.

Caution: The DataGrid is not editable or sortable if it is bound directly to a WebServiceConnector component or an XMLConnector component. You must bind the DataGrid component to the DataSet component and bind the DataSet component to the WebServiceConnector component or XMLConnector component if you want the grid to be editable or sortable. For more information, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Example

The following example prevents the first column in a grid from being edited:

```
myDataGrid.getColumnAt(0).editable = false;
```

See also

`DataGrid.editable`

DataGridColumn.headerRenderer

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnAt(index).headerRenderer
```

Description

Property; a string that indicates a class name to be used to display the header of this column. Any class used for this property must implement the CellRenderer API (see "CellRenderer API" on page 145). The default value is `undefined`.

Example

The following example uses a linkage identifier to set a new header renderer:

```
myGrid.getColumnAt(3).headerRenderer = "MyHeaderRenderer";
```

DataGridColumn.headerText

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnAt(index).headerText
```

Description

Property; the text in the column header. The default value is the column name.

This property allows you to display something other than the field name as the header.

Example

The following example sets the column header text to "The Price":

```
var myColumn = new DataGridColumn("price");
myColumn.headerText = "The Price";
```

DataGridColumn.labelFunction

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnAt(index).labelFunction
```

Description

Property; specifies a function to determine which field (or field combination) of each item to display. This function receives one parameter, *item*, which is the item being rendered, and must return a string representing the text to display. This property can be used to create virtual columns that have no equivalent field in the item.

Note: The specified function operates in a nondefined scope.

Example

The following example creates a virtual column:

```
var myCol = myGrid.addColumn("Subtotal");
myCol.labelFunction = function(item) {
  return "$" + (item.price + (item.price * salesTax));
};
```

DataGridColumn.resizable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnAt(index).resizable
```

Description

Property; a Boolean value that indicates whether a column can be resized by a user (`true`) or not (`false`). The `DataGrid.resizableColumns` property must be set to `true` for this property to take effect. The default value is `true`.

Example

The following example prevents the column at index 1 from being resized:

```
myGrid.getColumnAt(1).resizable = false;
```

DataGridColumn.sortable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnAt(index).sortable
```

Description

Property; a Boolean value that indicates whether a column can be sorted by a user (`true`) or not (`false`). The `DataGrid.sortableColumns` property must be set to `true` for this property to take effect. The default value is `true`.

Caution: The DataGrid is not editable or sortable if it is bound directly to a WebServiceConnector component or an XMLConnector component. You must bind the DataGrid component to the DataSet component and bind the DataSet component to the WebServiceConnector component or XMLConnector component if you want the grid to be editable or sortable. For more information, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Example

The following example prevents the column at index 1 from being sorted:

```
myGrid.getColumnAt(1).sortable = false;
```

DataGridColumn.sortOnHeaderRelease

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnAt(index).sortOnHeaderRelease
```

Description

Property; a Boolean value that indicates whether the column is sorted automatically (`true`) or not (`false`) when a user clicks a header. This property can be set to `true` only if `DataGridColumn.sortable` is set to `true`. If `DataGridColumn.sortOnHeaderRelease` is set to `false`, you can catch the `headerRelease` event and perform your own sort.

The default value is `true`.

Caution: The DataGrid is not editable or sortable if it is bound directly to a WebServiceConnector component or an XMLConnector component. You must bind the DataGrid component to the DataSet component and bind the DataSet component to the WebServiceConnector component or XMLConnector component if you want the grid to be editable or sortable. For more information, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Example

The following example lets you catch the `headerRelease` event to perform your own sort:

```
myGrid.getColumnAt(7).sortOnHeaderRelease = false;
```

DataGridColumn.width

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnAt(index).width
```

Description

Property; a number that indicates the width of the column, in pixels. The default value is 50.

Example

The following example makes a column half the default width:

```
myGrid.getColumnAt(4).width = 25;
```

DataHolder component (Flash Professional only)

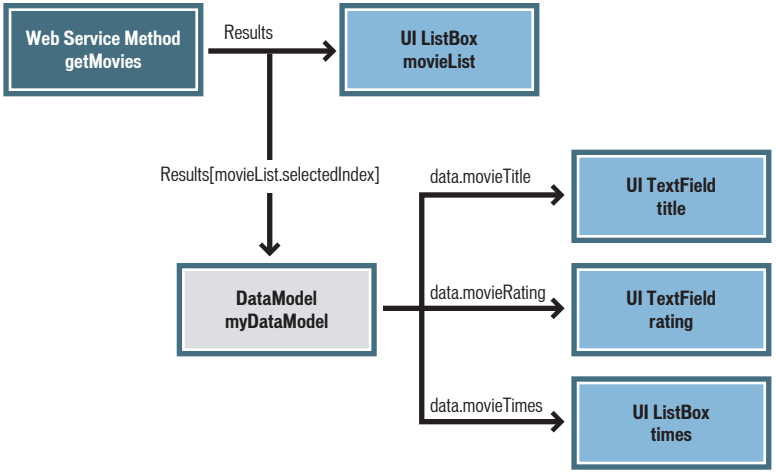
The DataHolder component is a repository for data and a means of generating events when that data has changed. Its main purpose is to hold data and act as a connector between other components that use data binding.

Initially, the DataHolder component has a single bindable property named `data`. You can add more properties by using the Schema tab in the Component inspector. For more information on using the Schema tab, see "Working with schemas in the Schema tab (Flash Professional only)" in *Using Flash*.

You can assign any type of data to a DataHolder property, either by creating a binding between the data and another property, or by using your own ActionScript code. Whenever the value of that data changes, the DataHolder component emits an event whose name is the same as the property, and any bindings associated with that property are executed.

In most cases, you will not use this component to build an application. It is needed only when you cannot bind external data directly to another component and you do not want to use a DataSet component. The DataHolder component is useful when you can't directly bind components (such as connectors, user interface components, or DataSet components) together. Below are some scenarios in which you might use a DataHolder component:

- If a data value is generated by ActionScript, you might want to bind it to some other components. In this case, you could have a DataHolder component that contains properties that are bound as desired. Whenever new values are assigned to those properties (by means of ActionScript, for example) those values will be distributed to the data-bound object.
- You might have a data value that results from a complex indexed data binding, as shown in the following diagram.



In this case it is convenient to bind the data value to a DataHolder component (called *DataModel* in this illustration) and then use that for bindings to the user interface.

Note: The DataHolder component is not meant to implement the same control over your data as the DataSet component. It does not manage or track data, nor does it have the ability to update data. It is a repository for holding data and generating events when that data has changed.

Creating an application with the DataHolder component (Flash Professional only)

In this example, you add an array property to a DataHolder component's schema (an array) whose value is determined by ActionScript code that you write. You then bind that array property to the `dataProvider` property of a DataGrid component by using the Bindings tab in the Component inspector.

To use the DataHolder component in a simple application:

1. In Flash MX Professional 2004, create a new file.
2. Open the Components panel, drag a DataHolder component to the Stage, and name it **dataHolder**.
3. Drag a DataGrid component to the Stage and name it **namesGrid**.
4. Select the DataHolder component and open the Component inspector.
5. Click the Schema tab in the Component inspector.
6. Click the Add Component Property (+) button located in the top pane of the Schema tab.
7. In the bottom pane of the Schema tab, type **namesArray** in the Field Name field, and select Array from the Data Type pop-up menu.
8. Click the Bindings tab in the Component inspector, and add a binding between the `namesArray` property of the DataHolder component and the `dataProvider` property of the DataGrid component.

   For more information on creating bindings with the Bindings tab, see "Working with bindings in the Bindings tab (Flash Professional only)" in *Using Flash*.
9. In the Timeline, select the first frame on Layer 1 and open the Actions panel.
10. Enter the following code in the Actions panel:

    ```
    dataHolder.namesArray= [{name:"Tim"},{name:"Paul"},{name:"Jason"}];
    ```

    This code populates the `namesArray` array with several objects. When this variable assignment executes, the binding that you established previously between the DataHolder component and the DataGrid component executes.
11. Test the file by selecting Control > Test Movie.

DataHolder class

Inheritance MovieClip > DataHolder

ActionScript class name mx.data.components.DataHolder

The DataHolder component is a repository for data and a means of generating events when that data has changed. Its main purpose is to hold data and act as a connector between other components that use data binding.

Initially, the DataHolder component has a single bindable property named `data`. You can add more properties by using the Schema tab in the Component inspector.

Property summary for the DataHolder class

The following table lists the properties of the DataHolder class.

Property	Description
`DataHolder.data`	Default bindable property for the DataHolder component.

DataHolder.data

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`dataHolder``.data`

Description

Property; the default item in a DataHolder object's schema. This property is not a "permanent" member of the DataHolder component. Rather, it is the default bindable property for each instance of the component. You can add your own bindable properties, or delete the default `data` property, by using the Schema tab in the Component inspector.

For more information on using the Schema tab, see "Working with schemas in the Schema tab (Flash Professional only)" in *Using Flash*.

Example

For a step-by-step example of using this component, see "Creating an application with the DataHolder component (Flash Professional only)" on page 287.

The following code shows a simple example of how to populate the DataHolder component with data that is a variable. To test the application, you enter a value into the text input field and click the `addDate_btn` instance, which adds the value to the DataHolder component. Click the `dumpDataHolder_btn` instance to trace the contents of the DataHolder component.

```
// Drag two Button components onto the Stage (addDate_btn and
  dumpDataHolder_btn), a TextInput (myDate_txt) and a DataHolder
  (myDataHolder). Add the following ActionScript to Frame 1:

var dhListener:Object = {};
dhListener.click = function() {
  trace("dumping DataHolder");
  trace("  "+myDataHolder.myDate);
  trace("");
};
var dateListener:Object = {};
dateListener.click = function() {
  myDataHolder.myDate = myDate_txt.text;
  trace("added value");
};
this.dumpDataHolder_btn.addEventListener("click", dhListener);
this.addDate_btn.addEventListener("click", dateListener);
```

DataProvider API

ActionScript Class Name mx.controls.listclasses.DataProvider

The DataProvider API is a set of methods and properties that a data source needs so that a list-based class can communicate with it. Arrays, recordsets, and data sets implement this API. You can create a DataProvider-compliant class by implementing all the methods and properties described in this section. A list-based component could then use that class as a data provider.

The methods of the DataProvider API let you query and modify the data in any component that displays data (also called a *view*). The DataProvider API also broadcasts `change` events when the data changes. Multiple views can use the same data provider and receive the `change` events.

A data provider is a linear collection (like an array) of items. Each item is an object composed of many fields of data. You can access these items by index (as you can with an array), using `DataProvider.getItemAt()`.

Data providers are most commonly used with arrays. Data-aware components apply all the methods of the DataProvider API to `Array.prototype` when an Array object is in the same frame or screen as a data-aware component. This lets you use any existing array as the data for views that have a `dataProvider` property.

Because of the DataProvider API, the version 2 components that provide views for data (DataGrid, List, Tree, and so on) can also display Flash Remoting RecordSet objects and data from the DataSet component. The DataProvider API is the language with which data-aware components communicate with their data providers.

In the Macromedia Flash documentation, "DataProvider" is the name of the API, `dataProvider` is a property of each component that acts as a view for data, and "data provider" is the generic term for a data source.

Method summary for the DataProvider API

The following table lists the methods of the DataProvider API.

Method	Description
`DataProvider.addItem()`	Adds an item at the end of the data provider.
`DataProvider.addItemAt()`	Adds an item to the data provider at the specified position.
`DataProvider.editField()`	Changes one field of the data provider.
`DataProvider.getEditingData()`	Gets the data for editing from a data provider.
`DataProvider.getItemAt()`	Gets a reference to the item at a specified position.
`DataProvider.getItemID()`	Returns the unique ID of the item.
`DataProvider.removeAll()`	Removes all items from a data provider.
`DataProvider.removeItemAt()`	Removes an item from a data provider at a specified position.
`DataProvider.replaceItemAt()`	Replaces the item at a specified position with another item.

Method	Description
DataProvider.sortItems()	Sorts the items in the data provider according to a compare function or sort options.
DataProvider.sortItemsBy()	Sorts the items in the data provider alphabetically or numerically, in the specified order, using the specified field name.

Property summary for the DataProvider API

The following table lists the properties of the DataProvider API.

Property	Description
DataProvider.length	The number of items in a data provider.

Event summary for the DataProvider API

The following table lists the events of the DataProvider API.

Event	Description
DataProvider.modelChanged	Broadcast when the data provider is changed.

DataProvider.addItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDP.addItem(*item*)

Parameters

item An object that contains data. This constitutes an item in a data provider.

Returns

Nothing.

Description

Method; adds a new item at the end of the data provider. This method triggers the `modelChanged` event with the event name `addItems`.

Example

The following example adds an item to the end of the data provider `myDP`:

```
myDP.addItem({label : "this is an Item"});
```

DataProvider.addItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDP.addItemAt(index, item)
```

Parameters

index A number greater than or equal to 0. This number indicates the position at which to insert the item; it is the index of the new item.

item An object containing the data for the item.

Returns

Nothing.

Description

Method; adds a new item to the data provider at the specified index. Indices greater than the data provider's length are ignored.

This method triggers the `modelChanged` event with the event name `addItems`.

Example

The following example adds an item to the data provider `myDP` at the fourth position:

```
myDP.addItemAt(3, {label : "this is the fourth Item"});
```

DataProvider.editField()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDP.editField(index, fieldName, newData)
```

Parameters

index A number greater than or equal to 0; the index of the item.

fieldName A string indicating the name of the field to modify in the item.

newData The new data to put in the data provider.

Returns

Nothing.

Description

Method; changes one field of the data provider.

This method triggers the `modelChanged` event with the event name `updateField`.

Example

The following code modifies the `label` field of the third item:

```
myDP.editField(2, "label", "mynewData");
```

DataProvider.getEditingData()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDP.getEditingData(index, fieldName)
```

Parameters

index A number greater than or equal to 0 and less than `DataProvider.length`. This number is the index of the item to retrieve.

fieldName A string indicating the name of the field being edited.

Returns

The editable formatted data to be used.

Description

Method; retrieves data for editing from a data provider. This lets the data model provide different formats of data for editing and displaying.

Example

The following code gets an editable string for the price field:

```
trace(myDP.getEditingData(4, "price");
```

DataProvider.getItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDP.getItemAt(*index*)

Parameters

index A number greater than or equal to 0 and less than DataProvider.length. This number is the index of the item to retrieve.

Returns

A reference to the retrieved item; undefined if the index is out of range.

Description

Method; retrieves a reference to the item at a specified position.

Example

The following code displays the label of the fifth item:

```
trace(myDP.getItemAt(4).label);
```

DataProvider.getItemID()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 Professional.

Usage

myDP.getItemID(*index*)

Parameters

index A number greater than or equal to 0.

Returns

A number that is the unique ID of the item.

Description

Method; returns a unique ID for the item. This method is primarily used to track selection. The ID is used in data-aware components to keep lists of what items are selected.

Example

This example gets the ID of the fourth item:

```
var ID = myDP.getItemID(3);
```

DataProvider.length

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDP.length
```

Description

Property (read-only); the number of items in the data provider.

Example

This example sends the number of items in the `myArray` data provider to the Output panel:

```
trace(myArray.length);
```

DataProvider.modelChanged

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.modelChanged = function(eventObject){
  // insert your code here
}
myMenu.addEventListener("modelChanged", listenerObject)
```

Description

Event; broadcast to all of its view listeners whenever the data provider is modified. You typically add a listener to a model by assigning its `dataProvider` property.

Version 2 components use a dispatcher/listener event model. When a data provider changes in some way, it broadcasts a `modelChanged` event, and data-aware components catch it to update their displays to reflect the changes in data.

The `Menu.modelChanged` event's event object has five additional properties:

- `eventName` The `eventName` property is used to subcategorize `modelChanged` events. Data-aware components use this information to avoid completely refreshing the component instance (view) that is using the data provider. The `eventName` property supports the following values:
  - `updateAll` The entire view needs refreshing, excluding scroll position.
  - `addItems` A series of items has been added.

- `removeItems` A series of items has been deleted.
- `updateItems` A series of items needs refreshing.
- `sort` The data has been sorted.
- `updateField` A field in an item must be changed and needs refreshing.
- `updateColumn` An entire field's definition in the data provider needs refreshing.
- `filterModel` The model has been filtered, and the view needs refreshing (reset the scroll position).
- `schemaLoaded` The field's definition of the data provider has been declared.

- `firstItem` The index of the first affected item.
- `lastItem` The index of the last affected item. The value equals `firstItem` if only one item is affected.
- `removedIDs` An array of the item identifiers that were removed.
- `fieldName` A string indicating the name of the field that is affected.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `listener` is defined and passed to `addEventListener()` as the second parameter. The event object is captured by the `modelChanged` handler in the `evt` parameter. When the `modelChanged` event is broadcast, a `trace` statement is sent to the Output panel.

```
listener = new Object();
listener.modelChanged = function(evt){
    trace(evt.eventName);
}
myList.addEventListener("modelChanged", listener);
```

DataProvider.removeAll()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDP.removeAll()
```

Parameters

None.

Returns

Nothing.

Description

Method; removes all items in the data provider. This method triggers the `modelChanged` event with the event name `removeItems`.

Example

This example removes all the items in the data provider:

```
myDP.removeAll();
```

DataProvider.removeItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDP.removeItemAt(index)
```

Parameters

index A number greater than or equal to 0. This number is the index of the item to remove.

Returns

Nothing.

Description

Method; removes the item at the specified index. The indices after the removed index collapse by one.

This method triggers the `modelChanged` event with the event name `removeItems`.

Example

This example removes the item at the fourth position:

```
myDP.removeItemAt(3);
```

DataProvider.replaceItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDP.replaceItemAt(index, item)
```

Parameters

index A number greater than or equal to 0. This number is the index of the item to change.

item An object that is the new item.

Returns

Nothing.

Description

Method; replaces the content of the item at the specified index. This method triggers the `modelChanged` event with the event name `removeItems`.

Example

This example replaces the item at index 3 with the item labeled "new label":

```
myDP.replaceItemAt(3, {label : "new label"});
```

DataProvider.sortItems()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myDP`.sortItems([`*compareFunc*`], [`*optionsFlag*`])`

Parameters

compareFunc A reference to a function that compares two items to determine their sort order. For details, see `Array.sort()` in *Flash ActionScript Language Reference*.This parameter is optional.

optionsFlag Lets you perform multiple, different types of sorts on a single array without having to replicate the entire array or resort it repeatedly. This parameter is optional.

The following are possible values for *optionsFlag*:

- `Array.DESCENDING`, which sorts highest to lowest.
- `Array.CASEINSENSITIVE`, which sorts case-insensitively.
- `Array.NUMERIC`, which sorts numerically if the two elements being compared are numbers. If they aren't numbers, use a string comparison (which can be case-insensitive if that flag is specified).
- `Array.UNIQUESORT`, which returns an error code (0) instead of a sorted array if two objects in the array are identical or have identical sort fields.

- `Array.RETURNINDEXEDARRAY`, which returns an integer index array that is the result of the sort. For example, the following array would return the second line of code and the array would remain unchanged:

```
["a", "d", "c", "b"]
[0, 3, 2, 1]
```

You can combine these options into one value. For example, the following code combines options 3 and 1:

```
array.sort (Array.NUMERIC | Array.DESCENDING)
```

Returns

Nothing.

Description

Method; sorts the items in the data provider according to the specified compare function or according to one or more specified sort options.

This method triggers the `modelChanged` event with the event name `sort`.

Example

This example sorts according to uppercase labels. The items `a` and `b` are passed to the function and contain `label` and `data` fields:

```
myList.sortItems(upperCaseFunc);
function upperCaseFunc(a,b){
    return a.label.toUpperCase() > b.label.toUpperCase();
}
```

DataProvider.sortItemsBy()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myDP`.sortItemsBy(`*fieldName*`, `*optionsFlag*`)`

myDP`.sortItemsBy(`*fieldName*`, `*order*`)`

Parameters

fieldName A string that specifies the name of the field to use for sorting. This value is usually `"label"` or `"data"`.

order A string that specifies whether to sort the items in ascending order (`"ASC"`) or descending order (`"DESC"`).

optionsFlag Lets you perform multiple, different types of sorts on a single array without having to replicate the entire array or resort it repeatedly. This parameter is optional.

The following are possible values for *optionsFlag*:

- `Array.DESCENDING`—sorts highest to lowest.
- `Array.CASEINSENSITIVE`—sorts case-insensitively.
- `Array.NUMERIC`—sorts numerically if the two elements being compared are numbers. If they aren't numbers, use a string comparison (which can be case-insensitive if that flag is specified).
- `Array.UNIQUESORT`—if two objects in the array are identical or have identical sort fields, this method returns an error code (0) instead of a sorted array.
- `Array.RETURNINDEXEDARRAY`—returns an integer index array that is the result of the sort. For example, the following array would return the second line of code and the array would remain unchanged:

```
["a", "d", "c", "b"]
[0, 3, 2, 1]
```

You can combine these options into one value. For example, the following code combines options 3 and 1:

```
array.sort (Array.NUMERIC | Array.DESCENDING)
```

Returns

Nothing.

Description

Method; sorts the items in the data provider in the specified order, using the specified field name. If the *fieldName* items are a combination of text strings and integers, the integer items are listed first. The *fieldName* parameter is usually `"label"` or `"data"`, but advanced programmers may specify any primitive value.

This method triggers the `modelChanged` event with the event name `sort`.

This is the fastest way to sort data in a component. It also maintains the component's selection state. The `sortItemsBy()` method is fast because it doesn't run any ActionScript while sorting. The `sortItems()` method needs to run an ActionScript compare function, and is therefore slower.

Example

The following code sorts the items in a list in ascending order using the labels of the list items:

```
myDP.sortItemsBy("label", "ASC");
```

DataSet component (Flash Professional only)

The DataSet component lets you work with data as collections of objects that can be indexed, sorted, searched, filtered, and modified.

The DataSet component functionality includes DataSetIterator, a set of methods for traversing and manipulating a data collection, and DeltaPacket, a set of interfaces and classes for working with updates to a data collection. In most cases, you don't use these classes and interfaces directly; you use them indirectly through methods provided by the DataSet class.

The items managed by the DataSet component are also called *transfer objects*. A transfer object exposes business data that resides on the server with public attributes or accessor methods for reading and writing data. The DataSet component allows developers to work with sophisticated client-side objects that mirror their server-side counterparts or, in its simplest form, a collection of anonymous objects with public attributes that represent the fields in a record of data. For details on transfer objects, see Core J2EE Patterns Transfer Object at http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html.

Note: The DataSet component requires Flash Player 7 or later.

Using the DataSet component

You typically use the DataSet component in combination with other components to manipulate and update a data source: a connector component for connecting to an external data source, user interface components for displaying data from the data source, and a resolver component for translating updates made to the data set into the appropriate format for sending to the external data source. You can then use data binding to bind properties of these different components together.

The DataSet component uses functionality in the data binding classes. If you intend to work with the DataSet component in ActionScript only, without using the Binding and Schema tabs in the Component inspector to set properties, you'll need to import the data binding classes into your FLA file and set required properties in your code. See "Making data binding classes available at runtime (Flash Professional only)" on page 213.

For general information on how to manage data in Flash using the DataSet component, see "Data management (Flash Professional only)" in *Using Flash*.

DataSet parameters

You can set the following parameters for the DataSet component:

itemClassName is a string indicating the name of the transfer object class that is instantiated each time a new item is created in the DataSet component.

The DataSet component uses transfer objects to represent the data that you retrieve from an external data source. If you leave this parameter blank, the data set creates an anonymous transfer object for you. If you give this parameter a value, the data set instantiates your transfer object whenever new data is added.

Note: You must make a fully qualified reference to this class somewhere in your code to make sure that it gets compiled into your application (such as `private var myItem:my.package.myItem;`).

logChanges is a Boolean value that defaults to `true`. If this parameter is set to `true`, the data set logs all changes made to its data and any method calls made on the associated transfer objects.

readOnly is a Boolean value that defaults to `false`. If this parameter is set to `true`, the data set cannot be modified.

You can write ActionScript to control these and additional options for the DataSet component using its properties, methods, and events. For more information, see "DataSet class (Flash Professional only)" on page 304.

Common workflow for the DataSet component

The typical workflow for the DataSet component is as follows.

To use a DataSet component:

1. Add an instance of the DataSet component to your application and give it an instance name.
2. Select the Schema tab for the DataSet component and create component properties to represent the persistent fields of the data set.
3. Load the DataSet component with data from an external data source. (For more information, see "About loading data into the DataSet component" in *Using Flash*.)
4. Use the Bindings tab of the Component inspector to bind the data set fields to UI components in your application.

   The UI controls are notified as records (transfer objects) are selected or modified within the DataSet component, and updated accordingly. In addition, the DataSet component is notified of changes made from within a UI control; those changes are tracked by the data set and can be extracted by means of a delta packet.
5. Call the methods of the DataSet component in your application to manage your data.

   Note: In addition to these steps, you can bind the DataSet component to a connector and a resolver component to provide a complete solution for accessing, managing, and updating data from an external data source.

Creating an application with the DataSet component

Typically, you use the DataSet component with other user interface components, and often with a connector component such as XMLConnector or WebServiceConnector. The items in the data set are populated by means of the connector component or raw ActionScript data, and then bound to user interface controls (such as List or DataGrid components).

The DataSet component uses functionality in the data binding classes. If you intend to work with the DataSet component in ActionScript only, without using the Binding and Schema tabs in the Component inspector to set properties, you'll need to import the data binding classes into your FLA file and set required properties in your code. See "Making data binding classes available at runtime (Flash Professional only)" on page 213.

To create an application using the DataSet component:

1. In Flash MX Professional 2004, select File > New. In the Type column, select Flash Document and click OK.
2. Open the Components panel if it's not already open.
3. Drag a DataSet component from the Components panel to the Stage. In the Property inspector, give it the instance name **userData**.
4. Drag a DataGrid component to the Stage and give it the instance name **userGrid**.
5. Resize the DataGrid component to be approximately 300 pixels wide and 100 pixels tall.
6. Drag a Button component to the Stage and set its instance name to **nextBtn**.
7. In the Timeline, select the first frame on Layer 1 and open the Actions panel.
8. Add the following code to the Actions panel:

```
var recData = [{id:0, firstName:"Mick", lastName:"Jones"},
               {id:1, firstName:"Joe", lastName:"Strummer"},
               {id:2, firstName:"Paul", lastName:"Simonon"}];
userData.items = recData;
```

   This populates the DataSet object's `items` property with an array of objects, each of which has three properties: `id`, `firstName`, and `lastName`.

9. Add the three properties and their required data types to the DataSet schema:
   a. Select the DataSet component on the Stage, open the Component inspector, and click the Schema tab.
   b. Click Add Component Property, and add three new properties, with field names `id`, `firstName`, and `lastName`, and data types `Number`, `String`, and `String`, respectively.

   Or, if you prefer to add the properties and their required data types in code, you can add the following line of code to the Actions panel instead of following steps a and b above:

```
// add required schema types
var i:mx.data.types.Str;
var j:mx.data.types.Num;
```

10. To bind the contents of the DataSet component to the contents of the DataGrid component, open the Component inspector and click the Bindings tab.
11. Select the DataGrid component (`userGrid`) on the Stage, and click the Add Binding (+) button in the Component inspector.
12. In the Add Binding dialog box, select "dataProvider : Array" and click OK.
13. Double-click the Bound To field in the Component inspector.
14. In the Bound To dialog box that appears, select "DataSet <userData>" from the Component Path column and then select "dataProvider : Array" from the Schema Location column.
15. To bind the selected index of the DataSet component to the selected index of the DataGrid component, select the DataGrid component on the Stage and click the Add Binding (+) button again in the Component inspector.
16. In the dialog box that appears, select "selectedIndex : Number". Click OK.

17. Double-click the Bound To field in the Component inspector to open the Bound To dialog box.
18. In the Component Path field, select "DataSet <userData>" from the Component Path column, and then select "selectedIndex : Number" from the Schema Location column.
19. Enter the following code in the Actions panel:

```
nextBtn.addEventListener("click", nextBtnClick);
function nextBtnClick(eventObj:Object):Void {
  userData.next();
}
```

    This code uses the `DataSet.next()` method to navigate to the next item in the DataSet object's collection of items. Since you had previously bound the `selectedIndex` property of the DataGrid object to the same property of the DataSet object, changing the current item in the DataSet object will change the current (selected) item in the DataGrid object, as well.
20. Save the file, and select Control > Test Movie to test the SWF file.

    The DataGrid object is populated with the specified items. Notice how clicking the button changes the selected item in the DataGrid object.

DataSet class (Flash Professional only)

Inheritance MovieClip > DataSet

ActionScript Class Name mx.data.components.DataSet

The DataSet component lets you work with data as collections of objects that can be indexed, sorted, searched, filtered, and modified.

The DataSet component functionality includes DataSetIterator, a set of methods for traversing and manipulating a data collection, and DeltaPacket, a set of interfaces and classes for working with updates to a data collection. In most cases, you don't use these classes and interfaces directly; you use them indirectly through methods provided by the DataSet class.

Method summary for the DataSet class

The following table lists the methods of the DataSet class.

Method	Description
`DataSet.addItem()`	Adds the specified item to the collection.
`DataSet.addItemAt()`	Adds an item to the data set at the specified position.
`DataSet.addSort()`	Creates a new sorted view of the items in the collection.
`DataSet.applyUpdates()`	Signals that the `deltaPacket` property has a value that you can access using data binding or ActionScript.
`DataSet.changesPending()`	Indicates whether the collection has changes pending that have not yet been sent in a delta packet.
`DataSet.clear()`	Clears all items from the current view of the collection.
`DataSet.createItem()`	Returns a newly initialized collection item.

Method	Description
DataSet.disableEvents()	Stops sending DataSet events to listeners.
DataSet.enableEvents()	Resumes sending DataSet events to listeners.
DataSet.find()	Locates an item in the current view of the collection.
DataSet.findFirst()	Locates the first occurrence of an item in the current view of the collection.
DataSet.findLast()	Locates the last occurrence of an item in the current view of the collection.
DataSet.first()	Moves to the first item in the current view of the collection.
DataSet.getItemId()	Returns the unique ID for the specified item.
DataSet.getIterator()	Returns a clone of the current iterator.
DataSet.getLength()	Returns the number of items in the data set.
DataSet.hasNext()	Indicates whether the current iterator is at the end of its view of the collection.
DataSet.hasPrevious()	Indicates whether the current iterator is at the beginning of its view of the collection.
DataSet.hasSort()	Indicates whether the specified sort exists.
DataSet.isEmpty()	Indicates whether the collection contains any items.
DataSet.last()	Moves to the last item in the current view of the collection.
DataSet.loadFromSharedObj()	Loads all of the relevant data needed to restore the DataSet collection from a shared object.
DataSet.locateById()	Moves the current iterator to the item with the specified ID.
DataSet.next()	Moves to the next item in the current view of the collection.
DataSet.previous()	Moves to the previous item in the current view of the collection.
DataSet.removeAll()	Removes all the items from the collection.
DataSet.removeItem()	Removes the specified item from the collection.
DataSet.removeItemAt()	Removes a data set item at a specified position.
DataSet.removeRange()	Removes the current iterator's range settings.
DataSet.removeSort()	Removes the specified sort from the DataSet object.
DataSet.saveToSharedObj()	Saves the data in the DataSet object to a shared object.
DataSet.setIterator()	Sets the current iterator for the DataSet object.
DataSet.setRange()	Sets the current iterator's range settings.
DataSet.skip()	Moves forward or backward by a specified number of items in the current view of the collection.
DataSet.useSort()	Makes the specified sort the active one.

Property summary for the DataSet class

The following table lists the properties of the DataSet class.

Property	Description
DataSet.currentItem	Returns the current item in the collection.
DataSet.dataProvider	Returns the data provider.
DataSet.deltaPacket	Returns changes made to the collection, or assigns changes to be made to the collection.
DataSet.filtered	Indicates whether items are filtered.
DataSet.filterFunc	User-defined function for filtering items in the collection.
DataSet.items	Items in the collection.
DataSet.itemClassName	Name of the object to create when assigning items.
DataSet.length	Specifies the number of items in the current view of the collection.
DataSet.logChanges	Indicates whether changes made to the collection, or its items, are recorded.
DataSet.properties	Contains the properties (fields) for any transfer object in this collection.
DataSet.readOnly	Indicates whether the collection can be modified.
DataSet.schema	Specifies the collection's schema in XML format.
DataSet.selectedIndex	Contains the current item's index in the collection.

Event summary for the DataSet class

The following table lists the events of the DataSet class.

Event	Description
DataSet.addItem	Broadcast before an item is added to the collection.
DataSet.afterLoaded	Broadcast after the `items` property is assigned.
DataSet.calcFields	Broadcast when calculated fields should be updated.
DataSet.deltaPacketChanged	Broadcast when the DataSet object's delta packet has been changed and is ready to be used.
DataSet.iteratorScrolled	Broadcast when the iterator's position is changed.
DataSet.modelChanged	Broadcast when items in the collection have been modified in some way.
DataSet.newItem	Broadcast when a new transfer object is constructed by the DataSet object, but before it is added to the collection.
DataSet.removeItem	Broadcast before an item is removed.
DataSet.resolveDelta	Broadcast when a delta packet is assigned to the DataSet object that contains messages.

DataSet.addItem

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(addItem) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.addItem = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("addItem", listenerObject)
```

Description

Event; generated just before a new record (transfer object) is inserted into this collection.

If you set the `result` property of the event object to `false`, the add operation is canceled; if you set it to `true`, the add operation is allowed.

The event object (*`eventObj`*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"addItem"`.

`item` A reference to the item in the collection to be added.

`result` A Boolean value that specifies whether the specified item should be added. By default, this value is `true`.

Example

The following `on(addItem)` event handler (attached to a DataSet object) cancels the addition of the new item if a user-defined function named `userHasAdminPrivs()` returns `false`; otherwise, the item addition is allowed.

```
on(addItem) {
  if(globalObj.userHasAdminPrivs()) {
    // Allow the item addition.
    eventObj.result = true;
  } else {
    // Don't allow item addition; user doesn't have admin privileges.
    eventObj.result = false;
  }
}
```

See also

`DataSet.removeItem`

DataSet.addItem()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.addItem([obj])
```

Parameters

obj An object to add to this collection. This parameter is optional.

Returns

A Boolean value: `true` if the item was added to the collection, `false` if it was not.

Description

Method; adds the specified record (transfer object) to the collection for management. The newly added item becomes the current item of the data set. If no *obj* parameter is specified, a new object is created automatically by means of `DataSet.createItem()`.

The location of the new item in the collection depends on whether a sort has been specified for the current iterator. If no sort is in use, the item is added to the end of the collection. If a sort is in use, the item is added to the collection according to its position in the current sort.

For more information on initialization and construction of the transfer object, see `DataSet.createItem()`.

Example

The following example uses `DataSet.createItem()` to create a new item and add it to the data set:

```
myDataSet.addItem(myDataSet.createItem());
```

See also

`DataSet.createItem()`

DataSet.addItemAt()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
DataSetInstance.addItemAt(index, item)
```

Parameters

index A number greater than or equal to 0. This number indicates the position at which to insert the item; it is the index of the new item.

item An object containing the data for the item.

Returns

A Boolean value indicating whether the item was added: `true` indicates that the item was added, and `false` indicates that the item already exists in the data set.

Description

Method; adds a new item to the data set at the specified index. Indices greater than the data provider's length are ignored.

This method triggers the `modelChanged` event with the event name `addItems`.

Example

The following example adds an item to the data set `myDataSet` at the fourth position:

```
myDataset.addItemAt(3, {label : "this is the fourth item"});
```

DataSet.addSort()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.addSort(name, fieldList, sortOptions)
```

Parameters

name A string that specifies the name of the sort.

fieldList An array of strings that specify the field names to sort on.

sortOptions One or more of the following integer (constant) values, which indicate what options are used for this sort. Separate multiple values using the bitwise OR operator (`|`). Specify one or more of the following values:

- `DataSetIterator.Ascending` Sorts items in ascending order. This is the default sort option, if none is specified.
- `DataSetIterator.Descending` Sorts items in descending order based on item properties specified.
- `DataSetIterator.Unique` Prevents the sort if any fields have like values.
- `DataSetIterator.CaseInsensitive` Ignores case when comparing two strings during the sort operation. By default, sorts are case-sensitive when the property being sorted on is a string.

A `DataSetError` exception is thrown when `DataSetIterator.Unique` is specified as a sort option and the data being sorted is not unique, when the specified sort name has already been added, or when a property specified in the *`fieldList`* array does not exist in this data set.

Returns

Nothing.

Description

Method; creates a new ascending or descending sort for the current iterator based on the properties specified by the *`fieldList`* parameter. Flash automatically assigns the new sort to the current iterator after it is created, and then stores it in the sorting collection for later retrieval.

Example

The following code creates a new sort named `"rank"` that performs a descending, case-sensitive, unique sort on the DataSet object's `"classRank"` field.

```
myDataSet.addSort("rank", ["classRank"], DataSetIterator.Descending |
  DataSetIterator.Unique | DataSetIterator.CaseInsensitive);
```

See also

`DataSet.removeSort()`

DataSet.afterLoaded

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(afterLoaded) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.afterLoaded = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("afterLoaded", listenerObject)
```

Description

Event; broadcast immediately after the `DataSet.items` property has been assigned.

The event object (*`eventObj`*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"afterLoaded"`.

Example

In this example, a form named `contactForm` (not shown) is made visible once the items in the data set `contact_ds` have been assigned.

```
contact_ds.addEventListener("afterLoaded", loadListener);
loadListener = new Object();
loadListener.afterLoaded = function (eventObj) {
  if(eventObj.target == "contact_ds") {
    contactForm.visible = true;
  }
}
```

DataSet.applyUpdates()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet`.applyUpdates()`

Parameters

None.

Returns

Nothing.

Description

Method; signals that the `DataSet.deltaPacket` property has a value that you can access using data binding or directly by ActionScript. Before this method is called, the `DataSet.deltaPacket` property is `null`. This method has no effect if events have been disabled by means of the `DataSet.disableEvents()` method.

Calling this method also creates a transaction ID for the current `DataSet.deltaPacket` property and emits a `deltaPacketChanged` event. For more information, see `DataSet.deltaPacket`.

Example

The following code calls the `applyUpdates()` method on `myDataSet`.

```
myDataSet.applyUpdates();
```

See also

`DataSet.deltaPacket`

DataSet.calcFields

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(calcFields) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.calcFields = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("calcFields", listenerObject)
```

Description

Event; generated when values of calculated fields for the current item in the collection need to be determined. A calculated field is one whose Kind property is set to Calculated on the Schema tab of the Component inspector. The `calcFields` event listener that you create should perform the required calculation and set the value for the calculated field.

This event is also called when the value of a noncalculated field (that is, a field with its Kind property set to Data on the Schema tab) is updated.

For more information on the Kind property, see "Schema kinds" in *Using Flash*.

Caution: Do not change the values of any of noncalculated fields in this event, because this will result in an "infinite loop." Set only the values of calculated fields within the `calcFields` event.

DataSet.changesPending()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.changesPending()
```

Parameters

None.

Returns

A Boolean value.

Description

Method; returns `true` if the collection, or any item in the collection, has changes pending that have not yet been sent in a delta packet; otherwise, returns `false`.

Example

The following code enables a Save Changes button (not shown) if the DataSet collection, or any items with that collection, have had modifications made to them that haven't been committed to a delta packet.

```
if(data_ds.changesPending()) {
  saveChanges_btn.enabled = true;
}
```

DataSet.clear()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet`.clear()`

Returns

Nothing.

Description

Method; removes the items in the current view of the collection. Which items are considered "viewable" depends on any current filter and range settings on the current iterator. Therefore, calling this method might not clear all of the items in the collection. To clear all of the items in the collection regardless of the current iterator's view, use `DataSet.removeAll()`.

If `DataSet.logChanges` is set to `true` when you invoke this method, "remove" entries are added to `DataSet.deltaPacket` for all items in the collection.

Example

This example removes all items from the current view of the DataSet collection. Because the `logChanges` property is set to `true`, the removal of those items is logged.

```
myDataSet.logChanges= true;
myDataSet.clear();
```

See also

`DataSet.deltaPacket`, `DataSet.logChanges`

DataSet.createItem()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.createItem([itemData])
```

Parameters

itemData Data associated with the item. This parameter is optional.

Returns

The newly constructed item.

Description

Method; creates an item that isn't associated with the collection. You can specify the class of object created by using the DataSet.itemClassName property. If no DataSet.itemClassName value is specified and the *itemData* parameter is omitted, an anonymous object is constructed. This anonymous object's properties are set to the default values based on the schema currently specified by DataSet.schema.

When this method is invoked, any listeners for the DataSet.newItem event are notified and are able to manipulate the item before it is returned by this method. The optional item data is used to initialize the class specified with the DataSet.itemClassName property or is used as the item if DataSet.itemClassName is blank.

A DataSetError exception is thrown when the class specified with the DataSet.itemClassName property cannot be loaded.

Example

```
contact.itemClassName = "Contact";
var itemData = new XML("<contact_info><name>John Smith</
  name><phone>555.555.4567</phone><zip><pre>94025</pre><post>0556</post></
  zip></contact_info>");
contact.addItem(contact.createItem(itemData));
```

See also

DataSet.itemClassName, DataSet.newItem, DataSet.schema

DataSet.currentItem

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.currentItem

Description

Property (read-only); returns the current item in the DataSet collection, or `null` if the collection is empty or if the current iterator's view of the collection is empty.

This property provides direct access to the item in the collection. Changes made by directly accessing this object are not tracked (in the `DataSet.deltaPacket` property), nor are any of the schema settings applied to any properties of this object.

Example

The following example displays the value of the `customerName` property defined in the current item in the data set named `customerData`.

```
trace(customerData.currentItem.customerName);
```

DataSet.dataProvider

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.dataProvider

Description

Property; the data provider for this data set. This property provides data to user interface controls, such as the List and DataGrid components.

For more information about the DataProvider API, see "DataProvider API" on page 290.

Example

The following code assigns the `dataProvider` property of a DataSet object to the corresponding property of a DataGrid component.

```
myGrid.dataProvider = myDataSet.dataProvider;
```

DataSet.deltaPacket

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.deltaPacket

Description

Property; returns a delta packet that contains all of the change operations made to the *dataSet* collection and its items. This property is `null` until `DataSet.applyUpdates()` is called on *dataSet*.

When `DataSet.applyUpdates()` is called, a transaction ID is assigned to the delta packet. This transaction ID is used to identify the delta packet on an update round trip from the server and back to the client. Any subsequent assignment to the `deltaPacket` property by a delta packet with a matching transaction ID is assumed to be the server's response to the changes previously sent. A delta packet with a matching ID is used to update the collection and report errors specified within the packet.

Errors or server messages are reported to listeners of the `DataSet.resolveDelta` event. Note that the `DataSet.logChanges` settings are ignored when a delta packet with a matching ID is assigned to `DataSet.deltaPacket`. A delta packet without a matching transaction ID updates the collection, as if the DataSet API were used directly. This may create additional delta entries, depending on the current `DataSet.logChanges` setting of *dataSet* and the delta packet.

A `DataSetError` exception is thrown if a delta packet is assigned with a matching transaction ID and one of the items in the newly assigned delta packet cannot be found in the original delta packet.

See also

`DataSet.applyUpdates()`, `DataSet.logChanges`, `DataSet.resolveDelta`

DataSet.deltaPacketChanged

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(deltaPacketChanged) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.deltaPacketChanged = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("deltaPacketChanged", listenerObject)
```

Description

Event; broadcast when the specified DataSet object's `deltaPacket` property has been changed and is ready to be used.

See also

`DataSet.deltaPacket`

DataSet.disableEvents()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.disableEvents()

Returns

Nothing.

Description

Method; disables events for the DataSet object. While events are disabled, no user interface controls (such as a DataGrid component) are updated when changes are made to items in the collection, or when the DataSet object is scrolled to another item in the collection.

To reenable events, you must call DataSet.enableEvents(). The disableEvents() method can be called multiple times, and enableEvents() must be called an equal number of times to reenable the dispatching of events.

Example

In this example, events are disabled before changes are made to items in the collection, so the DataSet object won't try to refresh controls and impact performance.

```
// Disable events for the data set
myDataSet.disableEvents();
myDataSet.last();
while(myDataSet.hasPrevious()) {
  var price = myDataSet.price;
  price = price * 0.5; // Everything's 50% off!
  myDataSet.price = price;
  myDataSet.previous();
}
// Tell the data set it's time to update the controls now
myDataSet.enableEvents();
```

See also

DataSet.enableEvents()

DataSet.enableEvents()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.enableEvents()

Returns

Nothing.

Description

Method; reenables events for the DataSet objects after events have been disabled by a call to DataSet.disableEvents(). To reenable events for the DataSet object, the enableEvents() method must be called an equal or greater number of times than disableEvents() was called.

Example

In this example, events are disabled before changes are made to items in the collection, so the DataSet object won't try to refresh controls and impact performance.

```
// Disable events for the data set
myDataSet.disableEvents();
myDataSet.last();
while(myDataSet.hasPrevious()) {
  var price = myDataSet.price;
  price = price * 0.5; // Everything's 50% off!
  myDataSet.price = price;
  myDataSet.previous();
}
// Tell the dataset it's time to update the controls now
myDataSet.enableEvents();
```

See also

DataSet.disableEvents()

DataSet.filtered

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.filtered

Description

Property; a Boolean value that indicates whether the data in the current iterator is filtered. The default value is false.When this property is true, the filter function specified by DataSet.filterFunc is called for each item in the collection.

Example

In the following example, filtering is enabled on the DataSet object named `employee_ds`. Suppose that each record in the DataSet collection contains a field named `empType`. The following filter function returns `true` if the `empType` field in the current item is set to `"management"`; otherwise, it returns `false`.

```
employee_ds.filtered = true;
employee_ds.filterFunc = function(item:Object) {
  // filter out employees who are managers...
  return(item.empType != "management");
}
```

See also

`DataSet.filterFunc`

DataSet.filterFunc

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.filterFunc = function(item:Object) {
  // return true|false;
};
```

Description

Property; specifies a function that determines which items are included in the current view of the collection. When `DataSet.filtered` is set to `true`, the function assigned to this property is called for each record (transfer object) in the collection. For each item that is passed to the function, it should return `true` if the item should be included in the current view, or `false` if the item should not be included in the current view.

When changing the filter function on a data set, you must set the `filtered` property to `false` and then `true` again in order for the proper `modelChanged` event to be generated. Changing the `filterFunc` property won't generate the event.

Also, if a filter is already in place when the data loads in (`modelChanged` or `updateAll`), the filter isn't applied until `filtered` is set to `false` and then back to `true` again.

Example

In the following example, filtering is enabled on the DataSet object named `employee_ds`. The specified filter function returns `true` if the `empType` field in each item is set to `"management"`; otherwise, it returns `false`.

```
employee_ds.filtered = true;
employee_ds.filterFunc = function(item:Object) {
  // filter out employees who are managers...
```

```
    return(item.empType != "management");
}
```

See also

DataSet.filtered

DataSet.find()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.find(*searchValues*)

Parameters

searchValues An array that contains one or more field values to be found within the current sort.

Returns

Returns `true` if the values are found; otherwise, returns `false`.

Description

Method; searches the current view of the collection for an item with the field values specified by *searchValues*. Which items are in the current view depends on any current filter and range settings. If an item is found, it becomes the current item in the DataSet object.

The values specified by *searchValues* must be in the same order as the field list specified by the current sort (see the example below).

If the current sort is not unique, the record (transfer object) found is nondeterministic. If you want to find the first or last occurrence of a transfer object in a nonunique sort, use DataSet.findFirst() or DataSet.findLast().

Conversion of the data specified is based on the underlying field's type. For example, if you specify `["05-02-02"]` as a search value, the underlying date field is used to convert the value using the date's `DataType.setAsString()` method. If you specify `[new Date().getTime()]`, the date's `DataType.setAsNumber()` method is used.

Example

This example searches for an item in the current collection whose `name` and `id` fields contain the values `"Bobby"` and 105, respectively. If found, DataSet.getItemId() is used to get the unique identifier for the item in the collection, and DataSet.locateById() is used to position the current iterator on that item.

```
var studentID:String = null;
studentData.addSort("id", ["name","id"]);
// Locate the transfer object identified by "Bobby" and 105.
```

```
// Note that the order of the search fields matches those
// specified in addSort().
if(studentData.find(["Bobby", 105])) {
  studentID = studentData.getItemId();
}
// Now use locateByID() to position the current iterator
// on the item in the collection whose ID matches studentID.
if(studentID != null) {
  studentData.locateById(studentID);
}
```

See also

DataSet.applyUpdates(), DataSet.getItemId(), DataSet.locateById()

DataSet.findFirst()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.findFirst(*searchValues*)

Parameters

searchValues An array that contains one or more field values to be found within the current sort.

Returns

Returns `true` if the items are found; otherwise, returns `false`.

Description

Method; searches the current view of the collection for the first item with the field values specified by *searchValues*. Which items are in the current view depends on any current filter and range settings.

The values specified by *searchValues* must be in the same order as the field list specified by the current sort (see the example below).

Conversion of the data specified is based on the underlying field's type. For example, if the search value specified is `["05-02-02"]`, the underlying date field is used to convert the value with the date's `setAsString()` method. If the value specified is `[new Date().getTime()]`, the date's `setAsNumber()` method is used.

Example

This example searches for the first item in the current collection whose `name` and `age` fields contain `"Bobby"` and `"13"`. If found, `DataSet.getItemId()` is used to get the unique identifier for the item in the collection, and `DataSet.locateById()` is used to position the current iterator on that item.

```
var studentID:String = null;
studentData.addSort("nameAndAge", ["name", "age"]);
// Locate the first transfer object with the specified values.
// Note that the order of the search fields matches those
// specified in addSort().
if(studentData.findFirst(["Bobby", "13"])) {
  studentID = studentData.getItemId();
}
// Now use locateByID() to position the current iterator
// on the item in the collection whose ID matches studentID.
if(studentID != null) {
  studentData.locateById(studentID);
}
```

See also

`DataSet.applyUpdates()`, `DataSet.getItemId()`, `DataSet.locateById()`

DataSet.findLast()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`dataSet``.findLast(`*`searchValues`*`)`

Parameters

`searchValues` An array that contains one or more field values to be found within the current sort.

Returns

Returns `true` if the items are found; otherwise, returns `false`.

Description

Method; searches the current view of the collection for the last item with the field values specified by *`searchValues`*. Which items are in the current view depends on any current filter and range settings.

The values specified by *`searchValues`* must be in the same order as the field list specified by the current sort (see the example below).

Conversion of the data specified is based on the underlying field's type. For example, if the search value specified is `["05-02-02"]`, the underlying date field is used to convert the value with the date's `setAsString()` method. If the value specified is `[new Date().getTime()]`, the date's `setAsNumber()` method is used.

Example

This example searches for the last item in the current collection whose `name` and `age` fields contain `"Bobby"` and `"13"`. If found, `DataSet.getItemId()` is used to get the unique identifier for the item in the collection, and `DataSet.locateById()` is used to position the current iterator on that item.

```
var studentID:String = null;
studentData.addSort("nameAndAge", ["name", "age"]);
// Locate the last transfer object with the specified values.
// Note that the order of the search fields matches those
// specified in addSort().
if(studentData.findLast(["Bobby", "13"])) {
  studentID = studentData.getItemId();
}
// Now use locateByID() to position the current iterator
// on the item in the collection whose ID matches studentID.
if(studentID != null) {
  studentData.locateById(studentID);
}
```

See also

`DataSet.applyUpdates()`, `DataSet.getItemId()`, `DataSet.locateById()`

DataSet.first()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`dataSet``.first()`

Returns

Nothing.

Description

Method; makes the first item in the current view of the collection the current item. Which items are in the current view depends on any current filter and range settings.

Example

The following code positions the data set `inventoryData` at the first item in its collection, and then displays the value of the `price` property contained by that item using the `DataSet.currentItem` property.

```
inventoryData.first();
trace("The price of the first item is:" + inventoryData.currentItem.price);
```

See also

`DataSet.last()`

DataSet.getItemId()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`dataSet``.getItemId([`*`index`*`])`

Parameters

`index` A number specifying the item in the current view for which to get the ID. This parameter is optional.

Returns

A string.

Description

Method; returns the identifier of the current item in the collection, or that of the item specified by *`index`*. This identifier is unique only in this collection and is assigned automatically by `DataSet.addItem()`.

Example

The following code gets the unique ID for the current item in the collection and then displays it in the Output panel.

```
var itemNo:String = myDataSet.getItemId();
trace("Employee id("+ itemNo+ ")");
```

See also

`DataSet.addItem()`

DataSet.getIterator()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.getIterator()
```

Returns

A ValueListIterator object.

Description

Method; returns a new iterator for this collection; this iterator is a clone of the current iterator in use, including its current position in the collection. This method is mainly for advanced users who want access to multiple, simultaneous views of the same collection.

Example

```
myIterator:ValueListIterator = myDataSet.getIterator();
myIterator.sortOn(["name"]);
myIterator.find({name:"John Smith"}).phone = "555-1212";
```

DataSet.getLength()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
DataSet.getLength()
```

Returns

The number of items in the data set.

Description

Method; returns the number of items in the data set.

Example

The following example calls `getLength()`:

```
//...
var myDSet:mx.data.components.DataSet;
myDSet = _parent.thisShelf.MyCompactDiscs;
trace ("Data set size is: " + myDSet.getLength());
//...
```

DataSet.hasNext()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.hasNext()
```

Returns

A Boolean value.

Description

Method; returns `false` if the current iterator is at the end of its view of the collection; otherwise, returns `true`.

Example

This example iterates over all of the items in the current view of the collection (starting at its beginning) and performs a calculation on the `price` property of each item.

```
myDataSet.first();
while(myDataSet.hasNext()) {
  var price = myDataSet.currentItem.price;
  price = price * 0.5; // Everything's 50% off!
  myDataSet.currentItem.price = price;
  myDataSet.next();
}
```

See also

`DataSet.currentItem`, `DataSet.first()`, `DataSet.next()`

DataSet.hasPrevious()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.hasPrevious()
```

Returns

A Boolean value.

Description

Method; returns `false` if the current iterator is at the beginning of its view of the collection; otherwise, returns `true`.

Example

This example iterates over all of the items in the current view of the collection (starting from the its last item) and performs a calculation on the `price` property of each item.

```
myDataSet.last();
while(myDataSet.hasPrevious()) {
  var price = myDataSet.currentItem.price;
  price = price * 0.5; // Everything's 50% off!
  myDataSet.currentItem.price = price;
  myDataSet.previous();
}
```

See also

DataSet.currentItem, DataSet.skip(), DataSet.previous()

DataSet.hasSort()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.hasSort(*sortName*)

Parameters

sortName A string that contains the name of a sort created with DataSet.addSort().

Returns

A Boolean value.

Description

Method; returns `true` if the sort specified by *sortName* exists; otherwise, returns `false`.

Example

The following code tests if a sort named "customerSort" exists. If the sort already exists, it is made the current sort by means of DataSet.useSort(). If a sort by that name doesn't exist, one is created by means of DataSet.addSort().

```
if(myDataSet.hasSort("customerSort")){
  myDataSet.useSort("customerSort");
} else {
  myDataSet.addSort("customerSort", ["customer"],
  DataSetIterator.Descending);
}
```

See also

DataSet.addSort(), DataSet.applyUpdates(), DataSet.useSort()

DataSet.isEmpty()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.isEmpty()

Returns

A Boolean value.

Description

Method; returns `true` if the specified DataSet object doesn't contain any items (that is, if *dataSet*.length == 0).

Example

The following code disables a Delete Record button (not shown) if the DataSet object it applies to is empty.

```
if(userData.isEmpty()){
    delete_btn.enabled = false;
}
```

See also

DataSet.length

DataSet.items

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myDataSet.items

Description

Property; an array of items managed by *myDataSet*.

Example

This example assigns an array of objects to a DataSet object's `items` property.

```
var recData = [{id:0, firstName:"Mick", lastName:"Jones"},
               {id:1, firstName:"Joe", lastName:"Strummer"},
               {id:2, firstName:"Paul", lastName:"Simonon"}];
myDataSet.items = recData;
```

DataSet.itemClassName

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.itemClassName
```

Description

Property; a string indicating the name of the class that should be created when items are added to the collection. The class you specify must implement the TransferObject interface, shown below.

```
interface mx.data.to.TransferObject {
  function clone():Object;
  function getPropertyData():Object;
  function setPropertyData(propData:Object):Void;
}
```

You can also set this property in the Property inspector.

To make the specified class available at runtime, you must also make a fully qualified reference to this class somewhere in your SWF file's code, as in the following code snippet:

```
var myItem:my.package.myItem;
```

A `DataSetError` exception is thrown if you try to modify the value of this property after the `DataSet.items` array has been loaded.

For more information, see "TransferObject interface" on page 756.

DataSet.iteratorScrolled

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(iteratorScrolled) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.iteratorScrolled = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("iteratorScrolled", listenerObject)
```

Description

Event; generated immediately after the current iterator has scrolled to a new item in the collection.

The event object (*eventObj*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"iteratorScrolled"`.

`scrolled` A number that specifies how many items the iterator scrolled; positive values indicate that the iterator moved forward in the collection; negative values indicate that it moved backward in the collection.

Example

In this example, the status bar of an application (not shown) is updated when the position of the current iterator changes.

```
on(iteratorScrolled) {
  var dataSet:mx.data.components.DataSet = eventObj.target;
  var statusBarText = dataSet.fullname+" Acct #:
  "+dataSet.getField("acctnum").getAsString();
  setStatusBar(statusBarText);
}
```

DataSet.last()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.last()
```

Returns

Nothing.

Description

Method; makes the last item in the current view of the collection the current item.

Example

The following code, attached to a Button component, goes to the last item in the DataSet collection.

```
function goLast(eventObj:obj) {
  inventoryData.last();
}
goLast_btn.addEventListener("click", goLast);
```

See also

DataSet.first()

DataSet.length

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.length

Description

Property (read-only); specifies the number of items in the current view of the collection. The viewable number of items is based on the current filter and range settings.

Example

The following example alerts users if they haven't made enough entries in the data set, perhaps using an editable DataGrid component.

```
if(myDataSet.length < MIN_REQUIRED) {
  alert("You need at least "+MIN_REQUIRED);
}
```

DataSet.loadFromSharedObj()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.loadFromSharedObj(*objName*, [*localPath*])

Parameters

objName A string specifying the name of the shared object to retrieve. The name can include forward slashes (for example, "work/addresses"). Spaces and the following characters are not allowed in the specified name:

~ % & \ ; : " ' , < > ? #

localPath An optional string parameter that specifies the full or partial path to the SWF file that created the shared object. This string is used to determine where the object is stored on the user's computer. The default value is the SWF file's full path.

Returns

Nothing.

Description

Method; loads all of the relevant data needed to restore this DataSet collection from a shared object. To save a DataSet collection to a shared object, use `DataSet.saveToSharedObj()`. The `DataSet.loadFromSharedObject()` method overwrites any data or pending changes that might exist in this DataSet collection. Note that the instance name of the DataSet collection is used to identify the data in the specified shared object.

This method throws a `DataSetError` exception if the specified shared object isn't found or if there is a problem retrieving the data from it.

Example

This example attempts to load a shared object named `webapp/customerInfo` associated with the data set named `myDataSet`. The method is called within a `try...catch` code block.

```
try {
  myDataSet.loadFromSharedObj("webapp/customerInfo");
}
catch(e:DataSetError) {
  trace("Unable to load shared object.");
}
```

See also

`DataSet.saveToSharedObj()`

DataSet.locateById()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`dataSet``.locateById(`*`id`*`)`

Parameters

`id` A string identifier for the item in the collection to be located.

Returns

A Boolean value.

Description

Method; positions the current iterator on the collection item whose ID matches *`id`*. This method returns `true` if the specified ID can be matched to an item in the collection; otherwise, it returns `false`.

Example

This example uses `DataSet.find()` to search for an item in the current collection whose `name` and `id` fields contain the values `"Bobby"` and 105, respectively. If found, `DataSet.getItemId()` is used to get the unique identifier for that item, and `DataSet.locateById()` is used to position the current iterator at that item.

```
var studentID:String = null;
studentData.addSort("id", ["name","id"]);
if(studentData.find(["Bobby", 105])) {
  studentID = studentData.getItemId();
  studentData.locateById(studentID);
}
```

See also

`DataSet.applyUpdates()`, `DataSet.find()`, `DataSet.getItemId()`

DataSet.logChanges

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`dataSet``.logChanges`

Description

Property; a Boolean value that specifies whether changes made to the data set, or its items, should (`true`) or should not (`false`) be recorded in `DataSet.deltaPacket`.

When this property is set to `true`, operations performed at the collection level and item level are logged. Collection-level changes include the addition and removal of items from the collection. Item-level changes include property changes made to items and method calls made on items by means of the DataSet component.

Example

The following example disables logging for the DataSet object named `userData`.

```
userData.logChanges = false;
```

See also

`DataSet.deltaPacket`

DataSet.modelChanged

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Description

```
on(modelChanged) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.modelChanged = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("modelChanged", listenerObject)
```

Description

Event; broadcast when the collection changes in some way—for example, when items are removed or added to the collection, when the value of an item's property changes, or when the collection is filtered or sorted.

The event object (*eventObj*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"iteratorScrolled"`.

`firstItem` The index (number) of the first item in the collection that was affected by the change.

`lastItem` The index (number) of the last item in the collection that was affected by the change (equals `firstItem` if only one item was affected).

`fieldName` A string that contains the name of the field being affected. This property is `undefined` unless the change was made to a property of the DataSet object.

`eventName` A string that describes the change that took place. This can be one of the following values:

String value	Description
`"addItems"`	A series of items has been added.
`"filterModel"`	The model has been filtered, and the view needs refreshing (reset scroll position).
`"removeItems"`	A series of items has been deleted.
`"schemaLoaded"`	The fields definition of the data provider has been declared.
`"sort"`	The data has been sorted.
`"updateAll"`	The entire view needs refreshing, excluding scroll position.
`"updateColumn"`	An entire field's definition in the data provider needs refreshing.

String value	Description
`"updateField"`	A field in an item has been changed and needs refreshing.
`"updateItems"`	A series of items needs refreshing.

Example

In this example, a Delete Item button is disabled if the items have been removed from the collection and the target DataSet object has no more items.

```
on(modelChanged) {
  delete_btn.enabled = ((eventObj.eventName == "removeItems") &&
  (eventObj.target.isEmpty()));
}
```

See also

`DataSet.isEmpty()`

DataSet.newItem

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(newItem) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.newItem = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("newItem", listenerObject)
```

Description

Event; broadcast when a new transfer object is constructed by means of `DataSet.createItem()`. A listener for this event can make modifications to the item before it is added to the collection.

The event object (*eventObj*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"iteratorScrolled"`.

`item` A referenece to the item that was created.

Example

This example makes modifications to a newly created item before it's added to the collection.

```
function newItemEvent(evt:Object):Void {
  var employee:Object = evt.item;
```

```
    employee.name = "newGuy";
    // property data happens to be XML
    employee.zip =
    employee.getPropertyData().firstChild.childNodes[1].attributes.zip;
}
employees_ds.addEventListener("newItem", newItemEvent);
```

DataSet.next()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.next()

Returns

Nothing.

Description

Method; makes the next item in the current view of the collection the current item. Which items are in the current view depends on any current filter and range settings.

Example

This example loops over all the items in a DataSet object, starting from the first item, and performs a calculation on a field in each item.

```
myDataSet.first();
while(myDataSet.hasNext()) {
  var price = myDataSet.price;
  price = price * 0.5; // Everything's 50% off!
  myDataSet.price = price;
  myDataSet.next();
}
```

See also

DataSet.first(), DataSet.hasNext()

DataSet.previous()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.previous()

Returns

Nothing.

Description

Method; makes the previous item in the current view of the collection the current item. Which items are in the current view depends on any current filter and range settings.

This example loops over all the items in the current view of the collection, starting from the last item, and performs a calculation on a field in each item.

```
myDataSet.last();
while(myDataSet.hasPrevious()) {
  var price = myDataSet.price;
  price = price * 0.5; // Everything's 50% off!
  myDataSet.price = price;
  myDataSet.previous();
}
```

See also

`DataSet.first()`, `DataSet.hasNext()`

DataSet.properties

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`dataSet``.properties`

Description

Property (read-only); returns an object that contains all of the exposed properties (fields) for any transfer object within this collection.

Example

This example displays all the names of the properties in the DataSet object named `myDataSet`.

```
for(var i in myDataSet.properties) {
  trace("field '"+i+ "' has value "+ myDataSet.properties[i]);
}
```

DataSet.readOnly

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.readOnly

Description

Property; a Boolean value that specifies whether this collection can be modified (`false`) or is read-only (`true`). Setting this property to `true` prevents updates to the collection. The default value is `false`.

You can also set this property in the Property inspector.

Example

The following example makes the DataSet object named `myDataSet` read-only, and then attempts to change the value of a property that belongs to the current item in the collection. This will throw an exception.

```
myDataSet.readOnly = true;
// This will throw an exception
myDataSet.currentItem.price = 15;
```

See also

DataSet.currentItem

DataSet.removeAll()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.removeAll()

Parameters

None.

Returns

Nothing.

Description

Method; removes all items in the DataSet collection.

Example

This example removes all the items in the DataSet collection `contact_ds`:

```
contact_ds.removeAll();
```

DataSet.removeItem

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(removeItem) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.removeItem = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("removeItem", listenerObject)
```

Description

Event; generated just before a new item is deleted from this collection.

If you set the `result` property of the event object to `false`, the delete operation is canceled; if you set it to `true`, the delete operation is allowed.

The event object (*`eventObj`*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"removeItem"`.

`item` A reference to the item in the collection to be removed.

`result` A Boolean value that specifies whether the item should be removed. By default, this value is `true`.

Example

In this example, an `on(removeItem)` event handler cancels the deletion of the new item if a user-defined function named `userHasAdminPrivs()` returns `false`; otherwise, the deletion is allowed.

```
on(removeItem) {
  if(globalObj.userHasAdminPrivs()) {
    // Allow the item deletion.
    eventObj.result = true;
  } else {
    // Don't allow item deletion; user doesn't have admin privileges.
    eventObj.result = false;
  }
}
```

See also

`DataSet.addItem`

DataSet.removeItem()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.removeItem([*item*])

Parameters

item The item to be removed. This parameter is optional.

Returns

A Boolean value. Returns `true` if the item was successfully removed; otherwise, returns `false`.

Description

Method; removes the specified item from the collection, or removes the current item if the *item* parameter is omitted. This operation is logged to `DataSet.deltaPacket` if `DataSet.logChanges` is `true`.

Example

The following code, attached to an instance of the Button component, removes the current item in the DataSet object named `usersData` that resides on the same Timeline as the Button instance.

```
on(click) {
  _parent.usersData.removeItem();
}
```

See also

`DataSet.deltaPacket`, `DataSet.logChanges`

DataSet.removeItemAt()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

DataSetInstance.removeItemAt(*index*)

Parameters

index A number greater than or equal to 0. This number is the index of the item to remove.

Returns

A Boolean value indicating whether the item was removed.

Description

Method; removes the item at the specified index. The indices after the removed index collapse by one.

This method triggers the `modelChanged` event with the event name `removeItems`.

In addition, it triggers the DataSet.removeItem event, which contains the `result` and `item` properties. The `result` property is used to determine if the item (referenced by the `item` property of the event) can be removed. By default, the `result` property is set to `true`, or if no event listener is specified for the `removeItem` event, the item will be removed by default.

An event listener can stop the item from being removed by listening for the `removeItem` event and setting the `result` property of the event to `false`, as shown in this example:

```
function removeItem(eventObj:Object):Void {
  // don't allow anyone to remove the item with customerId == 0
  eventObj.result = eventObj.item.customerId != 0;
}
```

Example

This example removes an item from the data set at the fourth position:

```
myDataSet.removeItemAt(3);
```

DataSet.removeRange()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.removeRange()
```

Returns

Nothing.

Description

Method; removes the current end point settings specified by `DataSet.setRange()` for the current iterator.

Example

```
myDataSet.addSort("name_id", ["name", "id"]);
myDataSet.setRange(["Bobby", 105],["Cathy", 110]);
while(myDataSet.hasNext()) {
  myDataSet.gradeLevel ="5"; // change all of the grades in this range
  myDataSet.next();
```

```
}
myDataSet.removeRange();
myDataSet.removeSort("name_id");
```

See also

DataSet.applyUpdates(), DataSet.hasNext(), DataSet.next(), DataSet.removeSort(), DataSet.setRange()

DataSet.removeSort()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.removeSort(*sortName*)

Parameters

sortName A string that specifies the name of the sort to remove.

Returns

Nothing.

Description

Method; removes the specified sort from this DataSet object if the sort exists. If the specified sort does not exist, this method throws a DataSetError exception.

Example

```
myDataSet.addSort("name_id", ["name", "id"]);
myDataSet.setRange(["Bobby", 105],["Cathy", 110]);
while(myDataSet.hasNext()) {
  myDataSet.gradeLevel ="5"; // change all of the grades in this range
  myDataSet.next();
}
myDataSet.removeRange();
myDataSet.removeSort("name_id");
```

See also

DataSet.applyUpdates(), DataSet.hasNext(), DataSet.next(), DataSet.removeRange(), DataSet.setRange()

DataSet.resolveDelta

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
on(resolveDelta) {
  // insert your code here
}
listenerObject = new Object();
listenerObject.resolveDelta = function (eventObj) {
  // insert your code here
}
dataSet.addEventListener("resolveDelta", listenerObject)
```

Description

Event; broadcast when `DataSet.deltaPacket` is assigned a delta packet whose transaction ID matches that of a delta packet previously retrieved from the DataSet object, and that has messages associated with any of the deltas or DeltaItem objects contained by that delta packet.

This event gives you the chance to reconcile any error returned from the server while attempting to apply changes previously submitted. Typically, you use this event to display a "reconcile dialog box" with the conflicting values, allowing the user to make appropriate modifications to the data so that it can be re-sent.

The event object (*eventObj*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"resolveDelta"`.

`data` An array of deltas and associated DeltaItem objects that have nonzero length messages.

Example

This example displays a form called `reconcileForm` (not shown) and calls a method on that form object (`setReconcileData()`) that allows the user to reconcile any conflicting values returned by the server.

```
myDataSet.addEventListener("resolveDelta", resolveDelta);
function resolveDelta(eventObj:Object) {
  reconcileForm.visible = true;
  reconcileForm.setReconcileData(eventObj.data);
}
// in the reconcileForm code
function setReconcileData(data:Array):Void {
  var di:DeltaItem;
  var ops:Array = ["property", "method"];
  var cl:Array;
  // change list
  var msg:String;
```

```
    for (var i = 0; i<data.length; i++) {
      cl = data[i].getChangeList();
      for (var j = 0; j<cl.length; j++) {
        di = cl[j];
        msg = di.getMessage();
        if (msg.length>0) {
          trace("The following problem occurred '"+msg+"' while performing a
'"+ops[di.kind]+"' modification on/with '"+di.name+"' current server value
["+di.curValue+"], value sent ["+di.newValue+"] Please fix!");
        }
      }
    }
}
```

DataSet.saveToSharedObj()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.saveToSharedObj(*objName*, [*localPath*])

Parameters

objName A string that specifies the name of the shared object to create. The name can include forward slashes (for example, "work/addresses"). Spaces and the following characters are not allowed in the specified name:

~ % & \ ; : " ' , < > ? #

localPath An optional string parameter that specifies the full or partial path to the SWF file that created the shared object. This string is used to determine where the object will be stored on the user's computer. The default value is the SWF file's full path.

Returns

Nothing.

Description

Method; saves all of the relevant data needed to restore this DataSet collection to a shared object. This allows users to work when disconnected from the source data, if it is a network resource. This method overwrites any data that might exist within the specified shared object for this DataSet collection. To restore a DataSet collection from a shared object, use DataSet.loadFromSharedObj(). Note that the instance name of the DataSet collection is used to identify the data within the specified shared object.

If the shared object can't be created or there is a problem flushing the data to it, this method throws a DataSetError exception.

Example

This example calls `saveToSharedObj()` in a `try..catch` block and displays an error if there is a problem saving the data to the shared object.

```
try {
  myDataSet.saveToSharedObj("webapp/customerInfo");
}
catch(e:DataSetError) {
  trace("Unable to create shared object”);
}
```

See also

`DataSet.loadFromSharedObj()`

DataSet.schema

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.schema

Description

Property; provides the XML representation of the schema for this DataSet object. The XML assigned to this property must have the following format:

```
<?xml version="1.0"?>
<properties>
  <property name="propertyName">
    <type name="dataType" />
    <encoder name="dataType">
      <options>
        <dataFormat>format options<dataFormat/>
      <options/>
    <encoder/>
    <kind name="dataKind">
    </kind>
  </property>
  <property> ... </property>
  ...
</properties>
```

A `DataSetError` exception is thrown if the XML specified does not follow the above format.

Example

The following example sets the schema of the data set `myDataSet` to a new XML object containing appropriately formatted XML:

```
myDataSet.schema = new XML("<properties><property name="billable"> ..etc.. </
  properties>");
```

DataSet.selectedIndex

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.selectedIndex

Description

Property; specifies the selected index in the collection. You can bind this property to the selected item in a DataGrid or List component, and vice versa. For a complete example that demonstrates this, see "Creating an application with the DataSet component" on page 302.

Example

The following example sets the selected index of a DataSet object (`userData`) to the selected index in a DataGrid component (`userGrid`).

```
userData.selectedIndex = userGrid.selectedIndex;
```

DataSet.setIterator()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.setIterator(*iterator*)

Parameters

iterator An iterator object returned by a call to `DataSet.getIterator()`.

Returns

Nothing.

Description

Method; assigns the specified iterator to this DataSet object and makes it the current iterator. The specified iterator must come from a previous call to `DataSet.getIterator()` on the DataSet object to which it is being assigned; otherwise; a `DataSetError` exception is thrown.

Example

```
myIterator:ValueListIterator = myDataSet.getIterator();
myIterator.sortOn(["name"]);
myDataSet.setIterator(myIterator);
```

See also

`DataSet.getIterator()`

DataSet.setRange()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`dataSet``.setRange(`*`startValues`*`, `*`endValues`*`)`

Parameters

`startValues` An array of key values of the properties of the first transfer object in the range.

`endValues` An array of key values of the properties of the last transfer object in the range.

Returns

Nothing.

Description

Method; sets the end points for the current iterator. The end points define a range in which the iterator operates. This is only valid if a valid sort has been set for the current iterator by means of `DataSet.applyUpdates()`.

Setting a range for the current iterator is more efficient than using a filter function if you want a grouping of values (see `DataSet.filterFunc`).

Example

```
myDataSet.addSort("name_id", ["name", "id"]);
myDataSet.setRange(["Bobby", 105],["Cathy", 110]);
while(myDataSet.hasNext()) {
  myDataSet.gradeLevel ="5"; // change all of the grades in this range
  myDataSet.next();
}
myDataSet.removeRange();
myDataSet.removeSort("name_id");
```

See also

DataSet.applyUpdates(), DataSet.hasNext(), DataSet.next(), DataSet.removeRange(), DataSet.removeSort()

DataSet.skip()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.skip(*offSet*)

Parameters

offSet An integer specifying the number of records by which to move the iterator position.

Returns

Nothing.

Description

Method; moves the current iterator's position forward or backward in the collection by the amount specified by *offSet*. Positive *offSet* values move the iterator's position forward; negative values move it backward.

If the specified offset is beyond the beginning (or end) of the collection, the iterator is positioned at the beginning (or end) of the collection.

Example

This example positions the current iterator at the first item in the collection, then moves to the next-to-last item and performs a calculation on a field belonging to that item.

```
myDataSet.first();
// Move to the item just before the last one
var itemsToSkip = myDataSet.length - 2;
myDataSet.skip(itemsToSkip).price = myDataSet.amount * 10;
```

DataSet.useSort()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

dataSet.useSort(*sortName*, *order*)

Parameters

sortName A string that contains the name of the sort to use.

order An integer value that indicates the sort order for the sort; the value must be `DataSetIterator.Ascending` or `DataSetIterator.Descending`.

Returns

Nothing.

Description

Method; switches the sort for the current iterator to the one specified by *sortName*, if it exists. If the specified sort does not exist, a `DataSetError` exception is thrown.

To create a sort, use `DataSet.applyUpdates()`.

Example

This code uses `DataSet.hasSort()` to determine if a sort named `"customer"` exists. If it does, the code calls `DataSet.useSort()` to make `"customer"` the current sort. Otherwise, the code creates a sort by that name using `DataSet.addSort()`.

```
if(myDataSet.hasSort("customer")) {
  myDataSet.useSort("customer");
} else {
  myDataSet.addSort("customer", ["customer"], DataSetIterator.Descending);
}
```

See also

`DataSet.applyUpdates()`, `DataSet.hasSort()`

DateChooser component (Flash Professional only)

The DateChooser component is a calendar that allows users to select a date. It has buttons that allow users to scroll through months and click a date to select it. You can set parameters that indicate the month and day names, the first day of the week, and disabled dates, as well as highlighting the current date.

A live preview of each DateChooser instance reflects the values indicated by the Property inspector or Component inspector during authoring.

Using the DateChooser component (Flash Professional only)

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

DateChooser parameters

You can set the following authoring parameters for each DateChooser component instance in the Property inspector or in the Component inspector:

monthNames sets the month names that are displayed in the heading row of the calendar. The value is an array and the default value is `["January", "February", "March", "April", "May", "June", "July", "August", "September", "October","November", "December"]`.

dayNames sets the names of the days of the week. The value is an array and the default value is `["S", "M", "T", "W", "T", "F", "S"]`.

firstDayOfWeek indicates which day of the week (0-6, 0 being the first element of the `dayNames` array) is displayed in the first column of the date chooser. This property changes the display order of the day columns.

disabledDays indicates the disabled days of the week. This parameter is an array and can have up to seven values. The default value is `[]` (an empty array).

showToday indicates whether to highlight today's date. The default value is `true`.

You can write ActionScript to control these and additional options for the DateChooser component using its properties, methods, and events. For more information, see "DateChooser class (Flash Professional only)" on page 354.

Creating an application with the DateChooser component

The following procedure explains how to add a DateChooser component to an application while authoring. In this example, the date chooser allows a user to pick a date for an airline reservation system. All dates before October 15th must be disabled. Also, a range in December must be disabled to create a holiday black-out period, and Mondays must be disabled.

To create an application with the DateChooser component:

1. Double-click the DateChooser component in the Components panel to add it to the Stage.
2. In the Property inspector, enter the instance name **flightCalendar**.
3. In the Actions panel, enter the following code on Frame 1 of the Timeline to set the range of selectable dates:

```
flightCalendar.selectableRange = {rangeStart:new Date(2003, 9, 15),
  rangeEnd:new Date(2003, 11, 31)}
```

   This code assigns a value to the `selectableRange` property in an ActionScript object that contains two Date objects with the variable names `rangeStart` and `rangeEnd`. This defines an upper and lower end of a range in which the user can select a date.
4. In the Actions panel, enter the following code on Frame 1 of the Timeline to set a range of holiday disabled dates:

```
flightCalendar.disabledRanges = [{rangeStart: new Date(2003, 11, 15),
  rangeEnd: new Date(2003, 11, 26)}];
```

5. In the Actions panel, enter the following code on Frame 1 of the Timeline to disable Mondays:

```
flightCalendar.disabledDays=[1];
```

6. Select Control > Test Movie.

Customizing the DateChooser component (Flash Professional only)

You can transform a DateChooser component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`).

Using styles with the DateChooser component

You can set style properties to change the appearance of a DateChooser instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

A DateChooser component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The glow color for the rollover and selected dates. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`backgroundColor`	Both	The background color. The default value is 0xEFEBEF (light gray).
`borderColor`	Both	The border color. The default value is 0x919999. The DateChooser component uses a solid single-pixel line as its border. This border cannot be modified through styles or skinning.

Style	Theme	Description
headerColor	Both	The background color for the component heading. The default color is white.
rollOverColor	Both	The background color of a rolled-over date. The default value is 0xE3FFD6 (bright green) with the Halo theme and 0xAAAAAA (light gray) with the Sample theme.
selectionColor	Both	The background color of the selected date. The default value is 0xCDFFC1 (light green) with the Halo theme and 0xEEEEEE (very light gray) with the Sample theme.
todayColor	Both	The background color for the today's date. The default value is 0x666666 (dark gray).
color	Both	The text color. The default value is 0x0B333C with the Halo theme and blank with the Sample theme.
disabledColor	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
embedFonts	Both	A Boolean value that indicates whether the font specified in fontFamily is an embedded font. This style must be set to true if fontFamily refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to true and fontFamily does not refer to an embedded font, no text will be displayed. The default value is false.
fontFamily	Both	The font name for text. The default value is "_sans".
fontSize	Both	The point size for the font. The default value is 10.
fontStyle	Both	The font style: either "normal" or "italic". The default value is "normal".
fontWeight	Both	The font weight: either "none" or "bold". The default value is "none". All components can also accept the value "normal" in place of "none" during a setStyle() call, but subsequent calls to getStyle() will return "none".
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".

The DateChooser component uses four categories of text to display the month name, the days of the week, today's date, and regular dates. The text style properties set on the DateChooser component itself control the regular date text and provide defaults for the other text. To set text styles for specific categories of text, use the following class-level style declarations.

Declaration name	Description
HeaderDateText	The month name.
WeekDayStyle	The days of the week.
TodayStyle	Today's date.

The following example demonstrates how to set the month name and days of the week to a deep red color.

```
_global.styles.HeaderDateText.setStyle("color", 0x660000);
_global.styles.WeekDayStyle.setStyle("color", 0x660000);
```

Using skins with the DateChooser component

The DateChooser component uses skins to represent the forward and back month buttons and the today indicator. To skin the DateChooser component while authoring, modify skin symbols in the Flash UI Components 2/Themes/MMDefault/DateChooser Assets/States folder in the library of one of the themes FLA files. For more information, see "About skinning components" on page 80.

Only the month scrolling buttons can be dynamically skinned in this component. A DateChooser component uses the following skin properties:

Property	Description
`backMonthButtonUpSymbolName`	The month back button up state. The default value is `backMonthUp`.
`backMonthButtonDownSymbolName`	The month back button pressed state. The default value is `backMonthDown`.
`backMonthButtonDisabledSymbolName`	The month back button disabled state. The default value is `backMonthDisabled`.
`fwdMonthButtonUpSymbolName`	The month forward button up state. The default value is `fwdMonthUp`.
`fwdMonthButtonDownSymbolName`	The month forward button pressed state. The default value is `fwdMonthDown`.
`fwdMonthButtonDisabledSymbolName`	The month forward button disabled state. The default value is `fwdMonthDisabled`.

The button symbols are used exactly as is without applying colors or resizing. The size is determined by the symbol during authoring.

To create movie clip symbols for DateChooser skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the DateChooser Assets folder to the library for your document.
4. Expand the DateChooser Assets/States folder in the library of your document.
5. Open the symbols you want to customize for editing.

   For example, open the backMonthDown symbol.

6. Customize the symbol as desired.

   For example, change the tint of the arrow to red.

7. Repeat steps 5-6 for all symbols you want to customize.

   For example, change the tint of the forward arrow down symbol to match the back arrow.

8. Click the Back button to return to the main Timeline.
9. Drag a DateChooser component to the Stage.
10. Select Control > Test Movie.

Note: The DateChooser Assets/States folder also has a Day Skins folder with a single skin element, `cal_todayIndicator`. This element can be modified during authoring to customize the today indicator. However, it cannot be changed dynamically on a particular DateChooser instance to use a different symbol. In addition, the `cal_todayIndicator` symbol must be a solid single-color graphic, because the DateChooser component will apply the `todayColor` color to the graphic as a whole. The graphic may have cut-outs, but keep in mind that the default text color for today's date is white and the default background for the DateChooser is white, so a cut-out in the middle of the today indicator skin element would make today's date unreadable unless either the background color or today text color is also changed.

DateChooser class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > DateChooser

ActionScript Class Name mx.controls.DateChooser

The properties of the DateChooser class let you access the selected date and the displayed month and year. You can also set the names of the days and months, indicate disabled dates and selectable dates, set the first day of the week, and indicate whether the current date should be highlighted.

Setting a property of the DateChooser class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.DateChooser.version);
```

Note: The code `trace(myDC.version);` returns `undefined`.

Method summary for the DateChooser class

There are no methods exclusive to the DateChooser class.

Methods inherited from the UIObject class

The following table lists the methods the DateChooser class inherits from the UIObject class. When calling these methods from the DateChooser object, use the form *dateChooserInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the DateChooser class inherits from the UIComponent class. When calling these methods from the DateChooser object, use the form *dateChooserInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the DateChooser class

The following table lists the properties that are exclusive to the DateChooser class.

Property	Description
DateChooser.dayNames	An array indicating the names of the days of the week.
DateChooser.disabledDays	An array indicating the days of the week that are disabled for all applicable dates in the date chooser.
DateChooser.disabledRanges	A range of disabled dates or a single disabled date.
DateChooser.displayedMonth	A number indicating an element in the `monthNames` array to display in the date chooser.
DateChooser.displayedYear	A number indicating the year to display.

Property	Description
DateChooser.firstDayOfWeek	A number indicating an element in the dayNames array to display in the first column of the date chooser.
DateChooser.monthNames	An array of strings indicating the month names.
DateChooser.selectableRange	A single selectable date or a range of selectable dates.
DateChooser.selectedDate	A Date object indicating the selected date.
DateChooser.showToday	A Boolean value indicating whether the current date is highlighted.

Properties inherited from the UIObject class

The following table lists the properties the DateChooser class inherits from the UIObject class. When accessing these properties from the DateChooser object, use the form *dateChooserInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the DateChooser class inherits from the UIComponent class. When accessing these properties from the DateChooser object, use the form *dateChooserInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the DateChooser class

The following table lists the events that are exclusive to the DateChooser class.

Event	Description
DateChooser.change	Broadcast when a date is selected.
DateChooser.scroll	Broadcast when the month buttons are clicked.

Events inherited from the UIObject class

The following table lists the events the DateChooser class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the DateChooser class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

DateChooser.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
on(change){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  ...
}
chooserInstance.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners when a date is selected.

The first usage example uses an `on()` handler and must be attached directly to a DateChooser instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the date chooser `myDC`, sends "_level0.myDC" to the Output panel:

```
on(change){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`chooserInstance`*) dispatches an event (in this case, `change`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a DateChooser instance called `myDC` is changed. The first line of code creates a listener object called `form`. The second line defines a function for the `change` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function, in this example `eventObj`, to generate a message. The `target` property of an event object is the component that generated the event (in this example, `myDC`). The `NumericStepper.maximum` property is accessed from the event object's `target` property. The last line calls `EventDispatcher.addEventListener()` from `myDC` and passes it the `change` event and the `form` listener object as parameters.

```
form.change = function(eventObj){
    trace("date selected " +  eventObj.target.selectedDate);
}
myDC.addEventListener("change", form);
```

DateChooser.dayNames

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.dayNames

Description

Property; an array containing the names of the days of the week. Sunday is the first day (at index position 0) and the rest of the day names follow in order. The default value is `["S", "M", "T", "W", "T", "F", "S"]`.

Example

The following example changes the value of the fifth day of the week (Thursday) from "T" to "R":

```
myDC.dayNames[4] = "R";
```

DateChooser.disabledDays

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.disabledDays

Description

Property; an array indicating the disabled days of the week. All the dates in a month that fall on the specified day are disabled. The elements of this array can have values from 0 (Sunday) to 6 (Saturday). The default value is `[]` (an empty array).

Example

The following example disables Sundays and Saturdays so that users can select only weekdays:

```
myDC.disabledDays = [0, 6];
```

DateChooser.disabledRanges

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.disabledRanges

Description

Property; disables a single day or a range of days. This property is an array of objects. Each object in the array must be either a Date object that specifies a single day to disable, or an object that contains either or both of the properties `rangeStart` and `rangeEnd`, each of whose value must be a Date object. The `rangeStart` and `rangeEnd` properties describe the boundaries of the date range. If either property is omitted, the range is unbounded in that direction.

The default value of `disabledRanges` is `undefined`.

Specify a full date when you define dates for the `disabledRanges` property. For example, specify `new Date(2003,6,24)` rather than `new Date()`. If you don't specify a full date, the time returns as `00:00:00`.

Example

The following example defines an array with `rangeStart` and `rangeEnd` Date objects that disable the dates between May 7 and June 7:

```
myDC.disabledRanges = [{rangeStart: new Date(2003, 4, 7), rangeEnd: new
  Date(2003, 5, 7)}];
```

The following example disables all dates after November 7:

```
myDC.disabledRanges = [ {rangeStart: new Date(2003, 10, 7)} ];
```

The following example disables all dates before October 7:

```
myDC.disabledRanges = [ {rangeEnd: new Date(2002, 9, 7)} ];
```

The following example disables only December 7:

```
myDC.disabledRanges = [ new Date(2003, 11, 7) ];
```

DateChooser.displayedMonth

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.displayedMonth

Description

Property; a number indicating which month is displayed. The number indicates an element in the `monthNames` array, with 0 being the first month. The default value is the month of the current date.

Example

The following example sets the displayed month to December:

```
myDC.displayedMonth = 11;
```

See also

`DateChooser.displayedYear`

DateChooser.displayedYear

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC`.displayedYear`

Description

Property; a four-digit number indicating which year is displayed. The default value is the current year.

Example

The following example sets the displayed year to 2010:

```
myDC.displayedYear = 2010;
```

See also

`DateChooser.displayedMonth`

DateChooser.firstDayOfWeek

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC`.firstDayOfWeek`

Description

Property; a number indicating which day of the week (0-6, 0 being the first element of the `dayNames` array) is displayed in the first column of the DateChooser component. Changing this property changes the order of the day columns but has no effect on the order of the `dayNames` property. The default value is 0 (Sunday).

Example

The following example sets the first day of the week to Monday:

```
myDC.firstDayOfWeek = 1;
```

See also

`DateChooser.dayNames`

DateChooser.monthNames

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.monthNames

Description

Property; an array of strings indicating the month names at the top of the DateChooser component. The default value is `["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"]`.

Example

The following example sets the month names for the instance `myDC`:

```
myDC.monthNames = ["Jan", "Feb","Mar","Apr", "May", "June","July", "Aug",
  "Sept","Oct", "Nov", "Dec"];
```

DateChooser.scroll

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
on(scroll){
```

```
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.scroll = function(eventObject){
  ...
}
myDC.addEventListener("scroll", listenerObject)
```

Description

Event; broadcast to all registered listeners when a month button is clicked.

The first usage example uses an `on()` handler and must be attached directly to a DateChooser instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the date chooser `myDC`, sends "_level0.myDC" to the Output panel:

```
on(scroll){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`myDC`*) dispatches an event (in this case, `scroll`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The scroll event's event object has an additional property, `detail`, that can have one of the following values: `nextMonth`, `previousMonth`, `nextYear`, `previousYear`.

Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a month button is clicked on a DateChooser instance called `myDC`. The first line of code creates a listener object called `form`. The second line defines a function for the `scroll` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function, in this example `eventObj`, to generate a message. The `target` property of an event object is the component that generated the event—in this example, `myDC`. The last line calls `EventDispatcher.addEventListener()` from `myDC` and passes it the `scroll` event and the `form` listener object as parameters.

```
form = new Object();
form.scroll = function(eventObj){
  trace(eventObj.detail);
```

```
}
myDC.addEventListener("scroll", form);
```

DateChooser.selectableRange

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.selectableRange

Description

Property; sets a single selectable date or a range of selectable dates. The user cannot scroll beyond the selectable range. The value of this property is an object that consists of two Date objects named rangeStart and rangeEnd. The rangeStart and rangeEnd properties designate the boundaries of the selectable date range. If only rangeStart is defined, all the dates after rangeStart are enabled. If only rangeEnd is defined, all the dates before rangeEnd are enabled. The default value is undefined.

If you want to enable only a single day, you can use a single Date object as the value of selectableRange.

Specify a full date when you define dates—for example, new Date(2003,6,24) rather than new Date(). If you don't specify a full date, the time returns as 00:00:00.

The value of DateChooser.selectedDate is set to undefined if it falls outside the selectable range.

The values of DateChooser.displayedMonth and DateChooser.displayedYear are set to the the nearest last month in the selectable range if the current month falls outside the selectable range. For example, if the current displayed month is August, and the selectable range is from June 2003 to July,2003, the displayed month will change to July 2003.

Example

The following example defines the selectable range as the dates between and including May 7 and June 7:

```
myDC.selectableRange = {rangeStart: new Date(2001, 4, 7), rangeEnd: new
  Date(2003, 5, 7)};
```

The following example defines the selectable range as the dates after and including May 7:

```
myDC.selectableRange = {rangeStart: new Date(2003, 4, 7)};
```

The following example defines the selectable range as the dates before and including June 7:

```
myDC.selectableRange = {rangeEnd: new Date(2003, 5, 7)};
```

The following example defines the selectable date as June 7 only:

```
myDC.selectableRange = new Date(2003, 5, 7);
```

DateChooser.selectedDate

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.selectedDate

Description

Property; a Date object that indicates the selected date if that value falls within the value of the `selectableRange` property. The default value is `undefined`.

You cannot set the `selectedDate` property within a disabled range, outside a selectable range, or on a day that has been disabled. If this property is set to one of these dates, the value is `undefined`.

Example

The following example sets the selected date to June 7:

```
myDC.selectedDate = new Date(2003, 5, 7);
```

DateChooser.showToday

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDC.showToday

Description

Property; a Boolean value that determines whether the current date is highlighted. The default value is `true`.

Example

The following example turns off the highlighting on today's date:

```
myDC.showToday = false;
```

DateField component (Flash Professional only)

The DateField component is a nonselectable text field that displays the date with a calendar icon on its right side. If no date has been selected, the text field is blank and the month of today's date is displayed in the date chooser. When a user clicks anywhere inside the bounding box of the date field, a date chooser pops up and displays the dates in the month of the selected date. When the date chooser is open, users can use the month scroll buttons to scroll through months and years and select a date. When a date is selected, the date chooser closes.

The live preview of the DateField does not reflect the values indicated by the Property inspector or Component inspector during authoring, because it is a pop-up component that is not visible during authoring.

Using the DateField component (Flash Professional only)

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

DateField parameters

You can set the following authoring parameters for each DateField component instance in the Property inspector or in the Component inspector:

monthNames sets the month names that are displayed in the heading row of the calendar. The value is an array and the default value is `["January", "February", "March", "April", "May", "June", "July", "August", "September", "October","November", "December"]`.

dayNames sets the names of the days of the week. The value is an array and the default value is `["S", "M", "T", "W", "T", "F", "S"]`.

firstDayOfWeek indicates which day of the week (0-6, 0 being the first element of `dayNames` array) is displayed in the first column of the date chooser. This property changes the display order of the day columns.

The default value is 0, which is "S".

disabledDays indicates the disabled days of the week. This parameter is an array and can have up to seven values. The default value is `[]` (an empty array).

showToday indicates whether to highlight today's date. The default value is `true`.

You can write ActionScript to control these and additional options for the DateField component using its properties, methods, and events. For more information, see "DateField class (Flash Professional only)" on page 371.

Creating an application with the DateField component

The following procedure explains how to add a DateField component to an application while authoring. In this example, the DateField component allows a user to pick a date for an airline reservation system. All dates before today's date must be disabled. Also, a 15-day range in December must be disabled to create a holiday black-out period. Also, some flights are not available on Mondays, so all Mondays must be disabled for those flights.

To create an application with the DateField component:

1. Double-click the DateField component in the Components panel to add it to the Stage.
2. In the Property inspector, enter the instance name **flightCalendar**.
3. In the Actions panel, enter the following code on Frame 1 of the Timeline to set the range of selectable dates:

```
flightCalendar.selectableRange = {rangeStart:new Date(2001, 9, 1),
  rangeEnd:new Date(2003, 11, 1)};
```

   This code assigns a value to the `selectableRange` property in an ActionScript object that contains two Date objects with the variable names `rangeStart` and `rangeEnd`. This defines an upper and lower end of a range within which the user can select a date.
4. In the Actions panel, enter the following code on Frame 1 of the Timeline to set the ranges of disabled dates, one during December, and one for all dates before the current date:

```
flightCalendar.disabledRanges = [{rangeStart: new Date(2003, 11, 15),
  rangeEnd: new Date(2003, 11, 31)}, {rangeEnd: new Date(2003, 6, 16)}];
```

5. In the Actions panel, enter the following code on Frame 1 of the Timeline to disable Mondays:

```
flightCalendar.disabledDays=[1];
```

6. Control > Test Movie.

Customizing the DateField component (Flash Professional only)

You can transform a DateField component horizontally while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`). Setting the width does not change the dimensions of the date chooser in the DateField component. However, you can use the `pullDown` property to access the DateChooser component and set its dimensions.

Using styles with the DateField component

You can set style properties to change the appearance of a date field instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

The DateField component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The glow color for the rollover and selected dates. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`
`backgroundColor`	Both	The background color. The default value is 0xEFEBEF (light gray).
`borderColor`	Both	The border color. The default value is 0x919999. The DateField component's drop-down list uses a solid single-pixel line as its border. This border cannot be modified through styles or skinning.
`headerColor`	Both	The background color for the drop-down heading. The default color is white.
`rollOverColor`	Both	The background color of a rolled-over date. The default value is 0xE3FFD6 (bright green) with the Halo theme and 0xAAAAAA (light gray) with the Sample theme.
`selectionColor`	Both	The background color of the selected date. The default value is a 0xCDFFC1 (light green) with the Halo theme and 0xEEEEEE (very light gray) with the Sample theme.
`todayColor`	Both	The background color for the today's date. The default value is 0x666666 (dark gray).
`color`	Both	The text color. The default value is 0x0B333C with the Halo theme and blank with the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.

The DateField component uses four categories of text to display the month name, the days of the week, today's date, and regular dates. The text style properties set on the DateField component itself control the regular date text and the text displayed in the collapsed state, and provide defaults for the other text. To set text styles for specific categories of text, use the following class-level style declarations.

Declaration name	Description
`HeaderDateText`	The month name.
`WeekDayStyle`	The days of the week.
`TodayStyle`	Today's date.

The following example demonstrates how to set the month name and days of the week to a deep red color.

```
_global.styles.HeaderDateText.setStyle("color", 0x660000);
_global.styles.WeekDayStyle.setStyle("color", 0x660000);
```

Using skins with the DateField component

The DateField component uses skins to represent the visual states of the pop-up icon, a RectBorder instance for the border around the text input, and a DateChooser instance for the pop-up. To skin the pop-up icon while authoring, modify skin symbols in the Flash UI Components 2/Themes/MMDefault/DateField Assets/States folder in the library of one of the themes FLA files. For more information, see "About skinning components" on page 80. For information about skinning the RectBorder and DateChooser instances, see "RectBorder class" on page 647 and "Using skins with the DateChooser component" on page 353.

Besides the skins used by the subcomponents mentioned above, a DateField component uses the following skin properties to dynamically skin the pop-up icon:

Property	Description
`openDateUp`	The up state of the pop-up icon.
`openDateDown`	The down state of the pop-up icon.
`openDateOver`	The over state of the pop-up icon.
`openDateDisabled`	The disabled state of the pop-up icon.

To create movie clip symbols for DateField skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the DateField Assets folder to the library for your document.
4. Expand the DateField Assets folder in the library of your document.

5. Ensure that the DateFieldAssets symbol is selected for Export in First Frame.
6. Expand the DateField Assets/States folder in the library of your document.
7. Open the symbols you want to customize for editing.

   For example, open the openIconUp symbol.
8. Customize the symbol as desired.

   For example, draw a down arrow over the calendar image.
9. Repeat steps 7-8 for all symbols you want to customize.

   For example, draw a down arrow over all of the symbols.
10. Click the Back button to return to the main Timeline.
11. Drag a DateField component to the Stage.
12. Select Control > Test Movie.

DateField class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > ComboBase > DateField

ActionScript Class Name mx.controls.DateField

The properties of the DateField class let you access the selected date and the displayed month and year. You can also set the names of the days and months, indicate disabled dates and selectable dates, set the first day of the week, and indicate whether the current date should be highlighted.

Setting a property of the DateField class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.DateField.version);
```

Note: The code `trace(myDateFieldInstance.version);` returns `undefined`.

Method summary for the DateField class

The following table lists methods of the DateField class.

Method	Description
`DateField.close()`	Closes the pop-up DateChooser subcomponent.
`DateField.open()`	Opens the pop-up DateChooser subcomponent.

Methods inherited from the UIObject class

The following table lists the methods the DateField class inherits from the UIObject class. When calling these methods from the DateField object, use the form *dateFieldInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the DateField class inherits from the UIComponent class. When calling these methods from the DateField object, use the form *dateFieldInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the DateField class

The following table lists properties of the DateField class.

Property	Description
DateField.dateFormatter	A function that formats the date to be displayed in the text field.
DateField.dayNames	An array indicating the names of the days of the week.
DateField.disabledDays	An array indicating the disabled days of the week.
DateField.disabledRanges	A range of disabled dates or a single disabled date.
DateField.displayedMonth	A number indicating which element in the `monthNames` array to display.
DateField.displayedYear	A number indicating the year to display.

Property	Description
DateField.firstDayOfWeek	A number indicating an element in the dayNames array to display in the first column of the DateField component.
DateField.monthNames	An array of strings indicating the month names.
DateField.pullDown	A reference to the DateChooser subcomponent. This property is read-only.
DateField.selectableRange	A single selectable date or a range of selectable dates.
DateField.selectedDate	A Date object indicating the selected date.
DateField.showToday	A Boolean value indicating whether the current date is highlighted.

Properties inherited from the UIObject class

The following table lists the properties the DateField class inherits from the UIObject class. When accessing these properties from the DateField object, use the form *dateFieldInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the DateField class inherits from the UIComponent class. When accessing these properties from the DateField object, use the form *dateFieldInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the DateField class

The following table lists events of the DateField class.

Event	Description
DateField.change	Broadcast when a date is selected.
DateField.close	Broadcast when the DateChooser subcomponent closes.
DateField.open	Broadcast when the DateChooser subcomponent opens.
DateField.scroll	Broadcast when the month buttons are clicked.

Events inherited from the UIObject class

The following table lists the events the DateField class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the DateField class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

DateField.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
on(change){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  ...
}
myDF.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners when a date is selected.

The first usage example uses an `on()` handler and must be attached directly to a DateField instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the date field `myDF`, sends "_level0.myDF" to the Output panel:

```
on(change){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*myDF*) dispatches an event (in this case, `change`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a date field called `myDF` is changed. The first line of code creates a listener object called `form`. The second line defines a function for the `change` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function, in this example `eventObj`, to generate a message. The `target` property of an event object is the component that generated the event—in this example, `myDF`. The `DateField.selectedDate` property is accessed from the event object's `target` property. The last line calls `EventDispatcher.addEventListener()` from `myDF` and passes it the `change` event and the `form` listener object as parameters.

```
function change(eventObj){
    trace("date selected " +  eventObj.target.selectedDate) ;
}
myDF.addEventListener("change", this);
```

DateField.close()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF`.close()`

Parameters

None.

Returns

Nothing.

Description

Method; closes the pop-up menu.

Example

The following code closes the date chooser pop-up of the `myDF` date field instance:

```
myDF.close();
```

DateField.close

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
on(close){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.close = function(eventObject){
   ...
}
myDF.addEventListener("close", listenerObject)
```

Description

Event; broadcast to all registered listeners when the DateChooser subcomponent closes after a user clicks outside the icon or selects a date.

The first usage example uses an `on()` handler and must be attached directly to a DateField instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the date field `myDF`, sends "_level0.myDF" to the Output panel:

```
on(close){
   trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`myDF`*) dispatches an event (in this case, `close`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when the date chooser in `myDF` closes. The first line of code creates a listener object called `form`. The second line defines a function for the `close` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function, in this example `eventObj`, to generate a message. The `target` property of an event object is the component that generated the event—in this example, `myDF`. The property is accessed from the event object's `target` property. The last line calls `EventDispatcher.addEventListener()` from `myDF` and passes it the `close` event and the `form` listener object as parameters.

```
form.close = function(eventObj){
  trace("PullDown Closed" + eventObj.target.selectedDate);
}
myDF.addEventListener("close", form);
```

DateField.dateFormatter

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.dateFormatter

Description

Property; a function that formats the date to be displayed in the text field. The function must receive a Date object as parameter, and return a string in the format to be displayed.

Example

The following example sets the function to return the format of the date to be displayed:

```
myDF.dateFormatter = function(d:Date){
  return d.getFullYear()+"/ "+(d.getMonth()+1)+"/ "+d.getDate();
};
```

DateField.dayNames

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.dayNames

Description

Property; an array containing the names of the days of the week. Sunday is the first day (at index position 0) and the other day names follow in order. The default value is `["S", "M", "T", "W", "T", "F", "S"]`.

Example

The following example changes the value of the fifth day of the week (Thursday) from "T" to "R":

```
myDF.dayNames[4] = "R";
```

DateField.disabledDays

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.disabledDays

Description

Property; an array indicating the disabled days of the week. All the dates in a month that fall on the specified day are disabled. The elements of this array can have values between 0 (Sunday) and 6 (Saturday). The default value is `[]` (an empty array).

Example

The following example disables Sundays and Saturdays so that users can select only weekdays:

```
myDF.disabledDays = [0, 6];
```

DateField.disabledRanges

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.disabledRanges

Description

Property; disables a single day or a range of days. This property is an array of objects. Each object in the array must be either a Date object specifying a single day to disable, or an object containing either or both of the properties `rangeStart` and `rangeEnd`, each of whose value must be a Date object. The `rangeStart` and `rangeEnd` properties describe the boundaries of the date range. If either property is omitted, the range is unbounded in that direction.

The default value of `disabledRanges` is `undefined`.

Specify a full date when you define dates for the `disabledRanges` property—for example, `new Date(2003,6,24)` rather than `new Date()`. If you don't specify a full date, the time returns as `00:00:00`.

Example

The following example defines an array with `rangeStart` and `rangeEnd` Date objects that disable the dates between May 7 and June 7:

```
myDF.disabledRanges = [ {rangeStart: new Date(2003, 4, 7), rangeEnd: new
  Date(2003, 5, 7)}];
```

The following example disables all dates after November 7:

```
myDF.disabledRanges = [ {rangeStart: new Date(2003, 10, 7)} ];
```

The following example disables all dates before October 7:

```
myDF.disabledRanges = [ {rangeEnd: new Date(2002, 9, 7)} ];
```

The following example disables only December 7:

```
myDF.disabledRanges = [ new Date(2003, 11, 7) ];
```

DateField.displayedMonth

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.displayedMonth

Description

Property; a number indicating which month is displayed. The number indicates an element in the `monthNames` array, with 0 being the first month. The default value is the month of the current date.

Example

The following example sets the displayed month to December:

```
myDF.displayedMonth = 11;
```

See also

DateField.displayedYear

DateField.displayedYear

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.displayedYear

Description

Property; a number indicating which year is displayed. The default value is the current year.

Example

The following example sets the displayed year to 2010:

```
myDF.displayedYear = 2010;
```

See also

DateField.displayedMonth

DateField.firstDayOfWeek

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.firstDayOfWeek

Description

Property; a number indicating which day of the week (0-6, 0 being the first element of the dayNames array) is displayed in the first column of the DateField component. Changing this property changes the order of the day columns but has no effect on the order of the dayNames property. The default value is 0 (Sunday).

Example

The following example sets the first day of the week to Monday:

```
myDF.firstDayOfWeek = 1;
```

See also

DateField.dayNames

DateField.monthNames

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.monthNames

Description

Property; an array of strings indicating the month names at the top of the DateField component. The default value is `["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"]`.

Example

The following example sets the month names for the instance `myDF`:

```
myDF.monthNames = ["Jan", "Feb","Mar","Apr", "May", "June","July", "Aug",
  "Sept","Oct", "Nov", "Dec"];
```

DateField.open()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myDF.open()
```

Parameters

None.

Returns

Nothing.

Description

Method; opens the pop-up DateChooser subcomponent.

Example

The following code opens the pop-up date chooser of the `df` instance:

```
df.open();
```

DateField.open

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
on(open){
```

```
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.open = function(eventObject){
  ...
}
myDF.addEventListener("open", listenerObject)
```

Description

Event; broadcast to all registered listeners when a DateChooser subcomponent opens after a user clicks on the icon.

The first usage example uses an `on()` handler and must be attached directly to a DateField instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the date field `myDF`, sends "_level0.myDF" to the Output panel:

```
on(open){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`myDF`*) dispatches an event (in this case, `open`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a date field called `myDF` is opened. The first line of code creates a listener object called `form`. The second line defines a function for the `open` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function, in this example `eventObj`, to generate a message. The `target` property of an event object is the component that generated the event—in this example, `myDF`. The `DateField.selectedDate` property is accessed from the event object's `target` property. The last line calls `EventDispatcher.addEventListener()` from `myDF` and passes it the `open` event and the `form` listener object as parameters.

```
form.open = function(eventObj){
    trace("Pop-up opened and date selected is " +
  eventObj.target.selectedDate) ;
```

```
}
myDF.addEventListener("open", form);
```

DateField.pullDown

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.pullDown

Description

Property (read-only); a reference to the DateChooser component contained by the DateField component. The DateChooser subcomponent is instantiated when a user clicks on the DateField component. However, if the `pullDown` property is referenced before the user clicks on the component, the DateChooser is instantiated and then hidden.

Example

The following example sets the visibility of the DateChooser subcomponent to `false` and then sets the size of the DateChooser subcomponent to 300 pixels high and 300 pixels wide:

```
myDF.pullDown._visible = false;
myDF.pullDown.setSize(300,300);
```

DateField.scroll

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
on(scroll){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.scroll = function(eventObject){
   ...
}
myDF.addEventListener("scroll", listenerObject)
```

Description

Event; broadcast to all registered listeners when a month button is clicked.

The first usage example uses an `on()` handler and must be attached directly to a DateField instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the date field `myDF`, sends "_level0.myDF" to the Output panel:

```
on(scroll){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`myDF`*) dispatches an event (in this case, `scroll`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The scroll event's event object has an additional property, `detail`, that can have one of the following values: `nextMonth`, `previousMonth`, `nextYear`, `previousYear`.

Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a user clicks a month button on a DateField instance called `myDF`. The first line of code creates a listener object called `form`. The second line defines a function for the `scroll` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function, in this example `eventObj`, to generate a message. The `target` property of an event object is the component that generated the event—in this example, `myDF`. The last line calls `EventDispatcher.addEventListener()` from `myDF` and passes it the `scroll` event and the `form` listener object as parameters.

```
form = new Object();
form.scroll = function(eventObj){
  trace(eventObj.detail);
}
myDF.addEventListener("scroll", form);
```

DateField.selectableRange

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.selectableRange

Description

Property; sets a single selectable date or a range of selectable dates. The value of this property is an object that consists of two Date objects named rangeStart and rangeEnd. The rangeStart and rangeEnd properties designate the boundaries of the selectable date range. If only rangeStart is defined, all the dates after rangeStart are enabled. If only rangeEnd is defined, all the dates before rangeEnd are enabled. The default value is undefined.

If you want to enable only a single day, you can use a single Date object as the value of selectableRange.

Specify a full date when you define dates—for example, new Date(2003,6,24) rather than new Date(). If you don't specify a full date, the time returns as 00:00:00.

The value of DateField.selectedDate is set to undefined if it falls outside the selectable range.

The values of DateField.displayedMonth and DateField.displayedYear are set to the nearest last month in the selectable range if the current month falls outside the selectable range. For example, if the current displayed month is August, and the selectable range is from June 2003 to July 2003, the displayed month will change to July 2003.

Example

The following example defines the selectable range as the dates between and including May 7 and June 7:

```
myDF.selectableRange = {rangeStart: new Date(2001, 4, 7), rangeEnd: new
  Date(2003, 5, 7)};
```

The following example defines the selectable range as the dates after and including May 7:

```
myDF.selectableRange = {rangeStart: new Date(2003, 4, 7)};
```

The following example defines the selectable range as the dates before and including June 7:

```
myDF.selectableRange = {rangeEnd: new Date(2003, 5, 7)};
```

The following example defines the selectable date as June 7 only:

```
myDF.selectableRange = new Date(2003, 5, 7);
```

DateField.selectedDate

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.selectedDate

Description

Property; a Date object that indicates the selected date if that value falls within the value of the selectableRange property. The default value is undefined.

Example

The following example sets the selected date to June 7:

```
myDF.selectedDate = new Date(2003, 5, 7);
```

DateField.showToday

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myDF.showToday

Description

Property; a Boolean value that determines whether the current date is highlighted. The default value is true.

Example

The following example turns off the highlighting on today's date:

```
myDF.showToday = false;
```

Delegate class

Inheritance Object > Delegate

ActionScript Class Name mx.utils.Delegate

The Delegate class lets you run a function in a specific scope. This class is provided so that you can dispatch the same event to two different functions (see "Delegating events to functions" on page 63), and so that you can call functions within the scope of the containing class.

When you pass a function as a parameter to `EventDispatcher.addEventListener()`, the function is invoked in the scope of the broadcaster component instance, not the object in which it is declared (see "Delegating the scope of a function" on page 65). You can call `Delegate.create()` to call the function within the scope of the declaring object.

Method summary for the Delegate class

The following table lists the method of the Delegate class.

Property	Description
`Delegate.create()`	A static method that allows you to run a function in a specific scope.

Delegate.create()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Delegate.create(scopeObject, function)
```

Parameters

`scopeObject` A reference to an object. This is the scope in which to run the function.

`function` A reference to a function.

Description

Method (static); allows you to delegate events to specific scopes and functions. Use the following syntax:

```
import mx.utils.Delegate;
compInstance.addEventListener("eventName", Delegate.create(scopeObject,
  function));
```

The *`scopeObject`* parameter specifies the scope in which the specified function is called.

Example

For examples of `Delegate.create()`, see "Delegating events" on page 63.

See also

`EventDispatcher.addEventListener()`

Delta interface (Flash Professional only)

ActionScript Interface Name mx.data.components.datasetclasses.Delta

The Delta interface provides access to the transfer object, collection, and transfer object-level changes. With this interface you can access the new and previous values in a transfer object. For example, if the delta packet was obtained from a data set, each delta would represent an added, edited, or deleted row.

The Delta interface also provides access to messages returned by the associated server-side process. For more information on client-server interactions, see "RDBMSResolver component (Flash Professional only)" on page 636.

Use the Delta interface to examine the delta packet before sending changes to the server and to review messages returned from server-side processing.

Method summary for the Delta interface

The following table lists the methods of the Delta interface.

Method	Description
`Delta.addDeltaItem()`	Adds the specified DeltaItem instance.
`Delta.getChangeList()`	Returns an array of changes made to the current item.
`Delta.getDeltaPacket()`	Returns the delta packet that contains the delta.
`Delta.getId()`	Returns the unique ID of the current item within the DeltaPacket collection.
`Delta.getItemByName()`	Returns the specified DeltaItem object.
`Delta.getMessage()`	Returns the message associated with the current item.
`Delta.getOperation()`	Returns the operation that was performed on the current item within the original collection.
`Delta.getSource()`	Returns the transfer object on which the changes were performed.

Delta.addDeltaItem()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

delta.addDeltaItem(*deltaitem*)

Parameters

deltaitem DeltaItem instance to add to this delta.

Returns

Nothing.

Description

Method; adds the specified DeltaItem instance. If the specified DeltaItem instance already exists, this method replaces it.

Example

The following example calls the `addDeltaItem()` method:

```
//...
var d:Delta = new DeltaImpl("ID1345678", curItem, DeltaPacketConsts.Added, "",
  false);
d.addDeltaItem(new DeltaItem(DeltaItem.Property, "ID", {oldValue:15,
  newValue:16}));
//...
```

Delta.getChangeList()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`delta``.getChangeList()`

Parameters

None.

Returns

An array of associated DeltaItem instances.

Description

Method; returns an array of associated DeltaItem instances. Each DeltaItem instance in the array describes a change made to the item.

Example

The following example calls the `getChangeList()` method.:

```
//...
case mx.data.components.datasetclasses.DeltaPacketConsts.Modified: {
  // dpDelta is a variable of type Delta.
  var changes:Array = dpDelta.getChangeList();
  for(var i:Number = 0; i<changes.length; i++) {
    // getChangeMessage is a user-defined method.
    changeMsg = _parent.getChangeMessage(changes[i]);
    trace(changeMsg);
```

```
    }
//...
```

Delta.getDeltaPacket()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
delta.getDeltaPacket()
```

Parameters

None.

Returns

The delta packet that contains this delta.

Description

Method; returns the delta packet that contains this delta. This method lets you write code that can handle delta packets generically at the delta level.

Example

The following example uses the `getDeltaPacket()` method to access the delta packet's data source:

```
while(dpCursor.hasNext()) {
  dpDelta = Delta(dpCursor.next());
  trace("DeltaPacket source is: " + dpDelta.getDeltaPacket().getSource());
}
```

Delta.getId()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
delta.getId()
```

Parameters

None.

Returns

An object; returns the unique ID of this item within the DeltaPacket collection.

Description

Method; returns a unique identifier for this item within the DeltaPacket collection. Use this ID in the source component for the delta packet to receive updates and make changes to items that the delta packet was generated from. For example, assuming that the DataSet component sends updates to a server and the server returns new key field values, this method allows the DataSet component to examine the resulting delta packet, find the original transfer object, and make the appropriate updates to it.

Example

The following example calls the `getId()` method:

```
while(dpCursor.hasNext()) {
  dpDelta = Delta(dpCursor.next());
  trace("id ["+dpDelta.getId()+"]");
}
```

Delta.getItemByName()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`delta``.getItemByName(`*`name`*`)`

Parameters

`name` A string that specifies the name of the property or method for the associated DeltaItem object.

Returns

The DeltaItem object specified by *`name`*. If no DeltaItem object is found that matches *`name`*, this method returns `null`.

Description

Method; returns the DeltaItem object specified by *`name`*. When method calls or property changes on a transfer object are needed by name, this method provides the most efficient access.

Example

The following example calls the `getItemByName()` method:

```
private function buildFieldTag(deltaObj:Delta, field:Object,
  isKey:Boolean):String {
  var chgItem:DeltaItem = deltaObj.getItemByName(field.name);
  var result:String= "<field name=\"" + field.name + "\" type=\"" +
  field.type.name + "\"";
  var oldValue:String;
  var newValue:String;
```

```
  if (deltaObj.getOperation() != DeltaPacketConsts.Added) {
    oldValue = (chgItem != null ? (chgItem.oldValue != null ?
  encodeFieldValue(field.name, chgItem.oldValue) : __nullValue) :
  encodeFieldValue(field.name, deltaObj.getSource()[field.name]));
    newValue = (chgItem.newValue != null ? encodeFieldValue(field.name,
  chgItem.newValue) : __nullValue);
    result+= " oldValue=\""  + oldValue + "\"";
    result+= chgItem != null ? " newValue=\"" + newValue + "\"" : "";
    result+= " key=\"" + isKey.toString() + "\" />";
  }
  else {
    result+= " newValue=\"" +encodeFieldValue(field.name,
  deltaObj.getSource()[field.name]) + "\"";
    result+= " key=\"" + isKey.toString() + "\" />";
  }
  return result;
}
```

Delta.getMessage()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

delta.getMessage()

Parameters

None.

Returns

A string; returns the message associated with *delta*.

Description

Method; returns the associated message for this delta. Typically this message is only populated if the delta packet has been returned from a server in response to attempted updates. For more information, see "RDBMSResolver component (Flash Professional only)" on page 636.

Example

The following example calls the getMessage() method:

```
//...
var dpi:Iterator = dp.getIterator();
var d:Delta;
while(dpi.hasNext()) {
  d= dpi.next();
  trace(d.getMessage());
}
//...
```

Delta.getOperation()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

delta.getOperation()

Parameters

None.

Returns

A number; returns the operation that was performed on the item within the original collection.

Description

Method; returns the operation that was performed on this item within the original collection. Valid values for this are DeltaPacketConsts.Added, DeltaPacketConsts.Removed, and DeltaPacketConsts.Modified.

You must either import mx.data.components.datasetclasses.DeltaPacketConsts or fully qualify each constant.

Example

The following example calls the getOperation() method:

```
while(dpCursor.hasNext()) {
  dpDelta = Delta(dpCursor.next());
  op=dpDelta.getOperation();
  trace("DeltaPacket source is: " + dpDelta.getDeltaPacket().getSource());
  switch(op) {
    case mx.data.components.datasetclasses.DeltaPacketConsts.Added:
      trace("***In case DeltaPacketConsts.Added ***");
    case mx.data.components.datasetclasses.DeltaPacketConsts.Modified: {
      trace("***In case DeltaPacketConsts.Modified ***");
  }
}
```

Delta.getSource()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

delta.getSource()

Parameters

None.

Returns

The transfer object on which the changes were performed.

Description

Method; returns the transfer object on which the changes were performed.

Example

The following example calls the `getSource()` method:

```
while(dpCursor.hasNext()) {
  dpDelta = Delta(dpCursor.next());
  op=dpDelta.getOperation();
  switch(op) {
    case mx.data.components.datasetclasses.DeltaPacketConsts.Modified: {
    // the original values are
    trace("Unmodified source is: ");
    var src = dpDelta.getDeltaPacket().getSource();
    for(var i in src){
      if(typeof(src[i]) != "function"){
        trace(i+"="+src[i]);
      }
    }
  }
}
```

DeltaItem class (Flash Professional only)

ActionScript Class Name mx.data.components.datasetclasses.DeltaItem

The DeltaItem class provides information about an individual operation performed on a transfer object. It indicates whether a change was made directly to a property of the transfer object or whether the change was made by a method call. It also provides the original state of properties on a transfer object. For example, if the source of the delta packet was a data set, the DeltaItem object contains information about any field that was edited.

In addition to the above, a DeltaItem object can contain server response information such as current value and a message.

Use the DeltaItem class when accessing the changes in a delta packet. To access these changes, use `DeltaPacket.getIterator()`, which returns an iterator of deltas. Each delta contains zero or more DeltaItem instances, which you can access through `Delta.getItemByName()` or `Delta.getChangeList()`.

Property summary for the DeltaItem class

The following table lists the properties of the DeltaItem class.

Property	Description
`DeltaItem.argList`	If a change is made through a method call, this is the array of values that were passed to the method. This property is read-only.
`DeltaItem.curValue`	If a change is made to a property, this is the current server value of the property. This property is read-only.
`DeltaItem.delta`	The associated delta for the DeltaItem object. This property is read-only.
`DeltaItem.kind`	The type of change.
`DeltaItem.message`	The server message associated with the DeltaItem object.
`DeltaItem.name`	The name of the property or method that changed. This property is read-only.
`DeltaItem.newValue`	If a change was made to a property, this is the new value of the property. This property is read-only.
`DeltaItem.oldValue`	If a change was made to a property, this is the old value of the property. This property is read-only.

DeltaItem.argList

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.argList

Description

Property (read-only); an array of values passed to the change method. This property applies only if the change's kind is DeltaItem.Method.

DeltaItem.curValue

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.curValue

Description

Property (read-only); an object containing the current property value on the server's copy of the transfer object. This property applies only if the change's kind is DeltaItem.Property, and the property is relevant only in a delta that has been returned from a server and is being applied to the data set for user resolution.

DeltaItem.delta

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.delta

Description

Property (read-only); a delta associated with the DeltaItem object. When a DeltaItem object is created, it is associated with a delta and adds itself to the delta's list of changes. This property provides a reference to the delta that this item belongs to.

DeltaItem.kind

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.kind

Description

Property; a number that indicates the type of change. Use the following constants to evaluate this property:

- `DeltaItem.Property` The change was made to a property on the transfer object.
- `DeltaItem.Method` The change was made through a method call on the transfer object.

DeltaItem.message

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.message

Description

Property; a string containing a server message associated with this DeltaItem object. This can be any message for the property or method call change attempted in the delta packet. This message is usually relevant only in a delta that has been returned from a server and is being applied to the DataSet for resolution.

DeltaItem.name

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.name

Description

Property (read-only); a string containing the name of the changed property (if the change's kind is `DeltaItem.Property`) or the name of the method that made the change (if the change's kind is `DeltaItem.Method`).

DeltaItem.newValue

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.newValue

Description

Property (read-only); an object containing the new value of the property. This property applies only if the change's kind is DeltaItem.Property.

DeltaItem.oldValue

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaitem.oldValue

Description

Property (read-only); an object containing the old value of the property. This property applies only if the change's kind is DeltaItem.Property.

DeltaPacket interface (Flash Professional only)

ActionScript Interface Name mx.data.components.datasetclasses.DeltaPacket

The DeltaPacket interface is provided by the `deltaPacket` property of the DataSet component, which is part of the data management functionality in Flash MX Professional 2004. (For more information, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*). Typically the delta packet is used internally by resolver components.The DeltaPacket interface and the related Delta interface and DeltaItem class let you manage changes made to the data. These components have no visual appearance at runtime.

A delta packet is an optimized set of instructions that describe all changes that have been made to the data in a data set. When the `DataSet.applyUpdates()` method is called, the DataSet component populates the `DataSet.deltaPacket` property. Typically, this property is connected (by data binding) to a resolver component such as RDBMSResolver. The resolver converts the delta packet into an update packet in the appropriate format.

Note: Unless you are writing your own custom resolver, it is unlikely you will ever need to know about or write code that accesses methods or properties of a delta packet.

A delta packet contains one or more deltas (see "Delta interface (Flash Professional only)" on page 390), and each delta contains zero or more delta items (see "DeltaItem class (Flash Professional only)" on page 397).

Method summary for the DeltaPacket interface

The following table lists the methods of the DeltaPacket interface.

Method	Description
`DeltaPacket.getConfigInfo()`	Returns configuration information that is specific to the implementation of the DeltaPacket interface.
`DeltaPacket.getIterator()`	Returns the iterator for the delta packet that iterates through the delta packet's list of deltas.
`DeltaPacket.getSource()`	Returns the source of the delta packet. This is the component that has exposed this delta packet.
`DeltaPacket.getTimestamp()`	Returns the date and time at which the delta packet was instantiated.
`DeltaPacket.getTransactionId()`	Returns the transaction ID for this delta packet.
`DeltaPacket.logChanges()`	Indicates whether the consumer of the delta packet should log the changes it specifies.

DeltaPacket.getConfigInfo()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaPacket.getConfigInfo(*info*)

Parameters

info Object; contains information specific to the implementation.

Returns

An object that contains information required for the specific DeltaPacket implementation.

Description

Method; returns configuration information that is specific to the implementation of the DeltaPacket interface. This method allows implementations of the DeltaPacket interface to access custom information.

Example

The following example calls the `getConfigInfo()` method:

```
// ...
new DeltaPacketImpl(source, getTransactionId(), null, logChanges(),
  getConfigInfo());
// ...
```

DeltaPacket.getIterator()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaPacket.getIterator()

Parameters

None.

Returns

An interface to the iterator for the DeltaPacket collection that iterates through the delta packet's list of deltas.

Description

Method; returns the iterator for the DeltaPacket collection. This provides a mechanism for looping through the changes in the delta packet. For a complete description of this interface, see "Iterator interface (Flash Professional only)" on page 441.

Example

The following example uses the `getIterator()` method to access the iterator for the deltas in a delta packet and uses a `while` statement to loop through the deltas:

```
var deltapkt:DeltaPacket = _parent.myDataSet.deltaPacket;
trace("*** Test deltapacket. Trans ID is: " + deltapkt.getTransactionId() + "
  ***");
var OPS:Array = new Array("added", "removed", "modified");
var dpCursor:Iterator = deltapkt.getIterator();
var dpDelta:Delta;
var op:Number;
var changeMsg:String;
while(dpCursor.hasNext()) {
  dpDelta = Delta(dpCursor.next());
  op=dpDelta.getOperation();
}
```

DeltaPacket.getSource()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaPacket`.getSource()`

Parameters

None.

Returns

An object; the source of the DeltaPacket collection. This object is typically a descendant of MovieClip, but this is not required. For example, if the source is a data set, this object might be `_level0.myDataSet`.

Description

Method; returns the source of the DeltaPacket collection.

Example

The following example calls the `getSource()` method:

```
// ...
var deltapkt:DeltaPacket = _parent.myDataSet.deltaPacket;
var dpSourceText:String = "Source: " + deltapkt.getSource();
trace(dpSourceText);
// ...
```

DeltaPacket.getTimestamp()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaPacket.getTimestamp()

Parameters

None.

Returns

A Date object containing the date and time at which the delta packet was created.

Description

Method; returns the date and time at which the delta packet was created.

Example

The following example calls the `getTimestamp()` method:

```
// ...
var deltapkt:DeltaPacket = _parent.myDataSet.deltaPacket;
var dpTime:String = " Time: " + deltapkt.getTimestamp();
trace(dpTime);
// ...
```

DeltaPacket.getTransactionId()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

deltaPacket.getTransactionId()

Parameters

None.

Returns

A string; the unique transaction ID for a single transaction grouping of delta packets.

Description

Method; returns the transaction ID for the delta packet. This unique identifier is used to group a send/receive transaction for a delta packet. The data set uses this to determine if the delta packet is part of the same transaction it originated with the `DataSet.applyUpdates()` call.

Example

The following example calls the `getTransactionId()` method:

```
// ...
var deltapkt:DeltaPacket = _parent.myDataSet.deltaPacket;
trace("*** Trans ID is: " + deltapkt.getTransactionId() + " ***");
// ...
```

DeltaPacket.logChanges()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`deltaPacket``.logChanges()`

Parameters

None.

Returns

A Boolean value; `true` if the consumer of the delta packet should log changes found in the delta packet.

Description

Method; returns `true` if the consumer of this delta packet should log the changes it specifies. This value is used mainly for communication of changes between data sets by means of shared objects or from a server to a local data set. In both cases, the data set should not record the changes specified.

Example

The following example calls the `logChanges()` method:

```
var deltapkt:DeltaPacket = _parent.myDataSet.deltaPacket;
if(deltapkt.logChanges()) {
  trace("*** We need to log changes. ***");
}
else {
  trace("*** We do not need to log changes");
}
```

DepthManager class

ActionScript Class Name mx.managers.DepthManager

The DepthManager class allows you manage the relative depth assignments of any component or movie clip, including `_root`. It also lets you manage reserved depths in a special highest-depth clip on `_root` for system-level services such as the cursor or tooltips.

In general, the Depth Manager manages components automatically. You do not need to use its APIs unless you are an advanced Flash developer.

The following methods constitute the relative depth-ordering API:

- `DepthManager.createChildAtDepth()`
- `DepthManager.createClassChildAtDepth()`
- `DepthManager.setDepthAbove()`
- `DepthManager.setDepthBelow()`
- `DepthManager.setDepthTo()`

The following methods constitute the reserved depth space API:

- `DepthManager.createClassObjectAtDepth()`
- `DepthManager.createObjectAtDepth()`

Method summary for the DepthManager class

The following table lists the methods of the DepthManager class.

Method	Description
`DepthManager.createChildAtDepth()`	Creates a child of the specified symbol at the specified depth.
`DepthManager.createClassChildAtDepth()`	Creates an object of the specified class at the specified depth.
`DepthManager.createClassObjectAtDepth()`	Creates an instance of the specified class at a specified depth in the special highest-depth clip.
`DepthManager.createObjectAtDepth()`	Creates an object at a specified depth in the highest-depth clip.
`DepthManager.setDepthAbove()`	Sets the depth above the specified instance.
`DepthManager.setDepthBelow()`	Sets the depth below the specified instance.
`DepthManager.setDepthTo()`	Sets the depth to the specified instance in the highest-depth clip.

Property summary for the DepthManager class

The following table lists the properties of the DepthManager class.

Property	Description
`DepthManager.kBottom`	A static property with the constant value 202.
`DepthManager.kCursor`	A static property with the constant value 101. This is the cursor depth.
`DepthManager.kNotopmost`	A static property with the constant value 204.
`DepthManager.kTooltip`	A static property with the constant value 102. This is the tooltip depth.
`DepthManager.kTop`	A static property with the constant value 201.
`DepthManager.kTopmost`	A static property with the constant value 203.

DepthManager.createChildAtDepth()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

movieClipInstance.createChildAtDepth(*linkageName*, *depthFlag*[, *initObj*])

Parameters

linkageName A linkage identifier. This parameter is a string.

depthFlag One of the following values: `DepthManager.kTop`, `DepthManager.kBottom`, `DepthManager.kTopmost`, `DepthManager.kNotopmost`. All depth flags are static properties of the DepthManger class. You must either reference the DepthManager package (for example, `mx.managers.DepthManager.kTopmost`), or use the `import` statement to import the DepthManager package.

initObj An initialization object. This parameter is optional.

Returns

A reference to the object created. The return type is MovieClip.

Description

Method; creates a child instance of the symbol specified by *linkageName* at the depth specified by *depthFlag*.

Example

The following example creates a `minuteHand` instance of the MinuteSymbol movie clip and places it in front of the clock:

```
import mx.managers.DepthManager;
minuteHand = clock.createChildAtDepth("MinuteSymbol", DepthManager.kTop);
```

DepthManager.createClassChildAtDepth()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
movieClipInstance.createClassChildAtDepth(className, depthFlag[, initObj])
```

Parameters

`className` A class name. This parameter is a of type Function.

`depthFlag` One of the following values: `DepthManager.kTop`, `DepthManager.kBottom`, `DepthManager.kTopmost`, `DepthManager.kNotopmost`. All depth flags are static properties of the DepthManger class. You must either reference the DepthManager package (for example, `mx.managers.DepthManager.kTopmost`), or use the `import` statement to import the DepthManager package.

`initObj` An initialization object. This parameter is optional.

Returns

A reference to the created child. The return type is UIObject.

Description

Method; creates a child of the class specified by *`className`* at the depth specified by *`depthFlag`*.

Example

The following code draws a focus rectangle in front of all NoTopmost objects:

```
import mx.managers.DepthManager
this.ring = createClassChildAtDepth(mx.skins.RectBorder, DepthManager.kTop);
```

The following code creates an instance of the Button class and passes it a value for its `label` property as an *`initObj`* parameter:

```
import mx.managers.DepthManager
button1 = createClassChildAtDepth(mx.controls.Button, DepthManager.kTop,
  {label: "Top Button"});
```

DepthManager.createClassObjectAtDepth()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
DepthManager.createClassObjectAtDepth(className, depthSpace[, initObj])
```

Parameters

className A class name. This parameter is of type Function.

depthSpace One of the following values: DepthManager.kCursor, DepthManager.kTooltip. All depth flags are static properties of the DepthManger class. You must either reference the DepthManager package (for example, mx.managers.DepthManager.kCursor), or use the import statement to import the DepthManager package.

initObj An initialization object. This parameter is optional.

Returns

A reference to the created object. The return type is UIObject.

Description

Method; creates an object of the class specified by *className* at the depth specified by *depthSpace*. This method is used for accessing the reserved depth spaces in the special highest-depth clip.

Example

The following example creates an object from the Button class:

```
import mx.managers.DepthManager
myCursorButton = createClassObjectAtDepth(mx.controls.Button,
  DepthManager.kCursor, {label: "Cursor"});
```

DepthManager.createObjectAtDepth()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
DepthManager.createObjectAtDepth(linkageName, depthSpace[, initObj])
```

Parameters

linkageName A linkage identifier. This parameter is of type String.

depthSpace One of the following values: DepthManager.kCursor, DepthManager.kTooltip. All depth flags are static properties of the DepthManger class. You must either reference the DepthManager package (for example, mx.managers.DepthManager.kCursor), or use the import statement to import the DepthManager package.

initObj An optional initialization object.

Returns

A reference to the created object. The return type is MovieClip.

Description

Method; creates an object at the specified depth. This method is used for accessing the reserved depth spaces in the special highest-depth clip.

Example

The following example creates an instance of the TooltipSymbol symbol and places it at the reserved depth for tooltips:

```
import mx.managers.DepthManager
myCursorTooltip = createObjectAtDepth("TooltipSymbol", DepthManager.kTooltip);
```

DepthManager.kBottom

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
DepthManager.kBottom
```

Description

Property (static); a property with the constant value 202. This property is passed as a parameter in calls to DepthManager.createClassChildAtDepth() and DepthManager.createChildAtDepth() to place content behind other content.

DepthManager.kCursor

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
DepthManager.kCursor
```

Description

Property (static); a property with the constant value 101. This property is passed as a parameter in calls to `DepthManager.createClassObjectAtDepth()` and `DepthManager.createObjectAtDepth()` to request placement at cursor depth.

DepthManager.kNotopmost

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
DepthManager.kNotopmost
```

Description

Property (static); a property with the constant value 204. This property is passed as a parameter in calls to `DepthManager.createClassChildAtDepth()` and `DepthManager.createChildAtDepth()` to request removal from the topmost layer.

DepthManager.kTooltip

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
DepthManager.kTooltip
```

Description

Property (static); a property with the constant value 102. This property is passed as a parameter in calls to `DepthManager.createClassObjectAtDepth()` and `DepthManager.createObjectAtDepth()` to place an object at the tooltip depth.

DepthManager.kTop

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
DepthManager.kTop
```

Description

Property (static); a property with the constant value 201. This property is passed as a parameter in calls to `DepthManager.createClassChildAtDepth()` and `DepthManager.createChildAtDepth()` to request placement on top of other content but below `DepthManager.kTopmost` content.

DepthManager.kTopmost

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
DepthManager.kTopmost
```

Description

Property (static); a property with the constant value 203. This property is used in calls to `DepthManager.createClassChildAtDepth()` and `DepthManager.createChildAtDepth()` to request placement on top of other content, including `DepthManager.kTop` objects.

DepthManager.setDepthAbove()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
movieClipInstance.setDepthAbove(instance)
```

Parameters

instance An instance name. This parameter is of type MovieClip.

Returns

Nothing.

Description

Method; sets the depth of a movie clip or component instance above the depth of the instance specified by the *instance* parameter and moves other objects if necessary.

DepthManager.setDepthBelow()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
movieClipInstance.setDepthBelow(instance)
```

Parameters

instance An instance name. This parameter is of type MovieClip.

Returns

Nothing.

Description

Method; sets the depth of a movie clip or component instance below the depth of the specified instance and moves other objects if necessary.

Example

The following code sets the depth of the `textInput` instance below the depth of `button`:

```
textInput.setDepthBelow(button);
```

DepthManager.setDepthTo()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
movieClipInstance.setDepthTo(depth)
```

Parameters

depth A depth level. This parameter is of type Number.

Returns

Nothing.

Description

Method; sets the depth of *movieClipInstance* to the value specified by *depth*. This method moves an instance to another depth to make room for another object.

Example

The following example sets the depth of the `mc1` instance to a depth of 10:

```
mc1.setDepthTo(10);
```

For more information about depth and stacking order, see "Determining the next highest available depth" in *Using ActionScript in Flash*.

EventDispatcher class

Events let your application know when the user has interacted with a component, and when important changes have occurred in the appearance or life cycle of a component—such as its creation, destruction, or resizing.

The methods of the EventDispatcher class let you add and remove event listeners so that your code can react to events appropriately. For example, you use the `EventDispatcher.addEventListener()` method to register a listener with a component instance. The listener is invoked when a component's event is triggered.

If you want to write a custom object that emits events that aren't related to the user interface, EventDispatcher is smaller and faster to use as a mix-in for UIComponent than UIEventDispatcher.

Event objects

An event object is passed to a listener as a parameter. The event object is an ActionScript object that has properties that contain information about the event that occurred. You can use the event object inside the listener callback function to access the name of the event that was broadcast, or the instance name of the component that broadcast the event. For example, the following code uses the `target` property of the `evtObj` event object to access the `label` property of the `myButton` instance and send the value to the Output panel:

```
listener = new Object();
listener.click = function(evtObj){
  trace("The " + evtObj.target.label + " button was clicked");
}
myButton.addEventListener("click", listener);
```

Some event object properties are defined in the W3C specification (www.w3.org/TR/DOM-Level-3-Events/events.html) but aren't implemented in version 2 of the Macromedia Component Architecture. Every version 2 event object has the properties listed in the table below. Some events have additional properties defined, and if so, the properties are listed in the event's entry.

Property	Description
`type`	A string indicating the name of the event.
`target`	A reference to the component instance broadcasting the event.

EventDispatcher class (API)

ActionScript Class Name mx.events.EventDispatcher

Method summary for the EventDispatcher class

The following table lists the methods of the EventDispatcher class.

Method	Description
`EventDispatcher.addEventListener()`	Registers a listener with a component instance.
`EventDispatcher.dispatchEvent()`	Dispatches an event programmatically.
`EventDispatcher.removeEventListener()`	Removes an event listener from a component instance.

EventDispatcher.addEventListener()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener(event, listener)
```

Parameters

event A string that is the name of the event.

listener A reference to a listener object or function.

Returns

Nothing.

Description

Method; registers a listener object with a component instance that is broadcasting an event. When the event occurs, the listener object or function is notified. You can call this method from any component instance. For example, the following code registers a listener to the component instance `myButton`:

```
myButton.addEventListener("click", myListener);
```

You must define the listener as either an object or a function before you call `addEventListener()` to register the listener with the component instance. If the listener is an object, it must have a callback function defined that is invoked when the event occurs. Usually, that callback function has the same name as the event with which the listener is registered. If the listener is a function, the function is invoked when the event occurs. For more information, see "Using listeners to handle events" on page 56.

You can register multiple listeners to a single component instance, but you must use a separate call to `addEventListener()` for each listener. Also, you can register one listener to multiple component instances, but you must use a separate call to `addEventListener()` for each instance. For example, the following code defines one listener object and assigns it to two Button component instances, whose `label` properties are `button1` and `button2`, respectively:

```
lo = new Object();
lo.click = function(evt){
  trace(evt.target.label + " clicked");
}
button1.addEventListener("click", lo);
button2.addEventListener("click", lo);
```

Execution order is not guaranteed. You cannot expect one listener to be called before another.

An event object is passed to the listener as a parameter. The event object has properties that contain information about the event that occurred. You can use the event object inside the listener callback function to access information about the type of event that occurred and which instance broadcast the event. In the example above, the event object is `evt` (you can use any identifier as the event object name), and it is used in the `if` statements to determine which button instance was clicked. For more information, see "About the event object" on page 66.

Example

The following example defines a listener object, `myListener`, and defines the callback function for the `click` event. It then calls `addEventListener()` to register the `myListener` listener object with the component instance `myButton`.

```
myListener = new Object();
myListener.click = function(evt){
  trace(evt.type + " triggered");
}
myButton.addEventListener("click", myListener);
```

To test this code, place a Button component on the Stage with the instance name `myButton`, and place this code in Frame 1.

EventDispatcher.dispatchEvent()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
dispatchEvent(eventObject)
```

Parameters

eventObject A reference to an event object. The event object must have a `type` property that is a string indicating the name of the event. Generally, the event object also has a `target` property that is the name of the instance broadcasting the event. You can define other properties on the event object that will help a user capture information about the event when it is dispatched.

Returns

Nothing.

Description

Method; dispatches an event to any listener registered with an instance of the class. This method is usually called from within a component's class file. For example, the SimpleButton.as class file dispatches the `click` event.

Example

The following example dispatches a `click` event:

```
dispatchEvent({type:"click"});
```

EventDispatcher.removeEventListener()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

componentInstance.removeEventListener(*event*, *listener*)

Parameters

event A string that is the name of the event.

listener A reference to a listener object or function.

Returns

Nothing.

Description

Method; unregisters a listener object from a component instance that is broadcasting an event.

FocusManager class

You can use the Focus Manager to specify the order in which components receive focus when a user presses the Tab key to navigate in an application. You can also use the Focus Manager to set a button in your document that receives keyboard input when a user presses Enter (Windows) or Return (Macintosh). For example, when users fill out a form, they should be able to tab between fields and press Enter (Windows) or Return (Macintosh) to submit the form.

All components implement Focus Manager support; you don't need to write code to invoke it.

The Focus Manager interacts with the System Manager, which activates and deactivates FocusManager instances as pop-up windows are activated or deactivated. Each modal window has an instance of FocusManager so the components in that window become their own tab set, preventing the user from tabbing into components in other windows.

The Focus Manager recognizes groups of radio buttons (those with a defined `RadioButton.groupName` property) and sets focus to the instance in the group that has a `selected` property that is set to `true`. When the Tab key is pressed, the Focus Manager checks to see if the next object has the same group name as the current object. If it does, it automatically moves focus to the next object with a different group name. Other sets of components that support a `groupName` property can also use this feature.

The Focus Manager handles focus changes caused by mouse clicks. If the user clicks a component, that component is given focus.

Using the Focus Manager

The Focus Manager does not automatically assign focus to a component. You must write a script that calls `FocusManager.setFocus()` on a component if you want a component to have focus when an application loads.

Note: If you call `FocusManager.setFocus()` to set focus to a component when an application loads, the focus ring does not appear around that component. The component has focus, but the indicator is not present.

To create focus navigation in an application, set the `tabIndex` property on any objects (including buttons) that should receive focus. When a user presses the Tab key, the Focus Manager looks for an enabled object with a `tabIndex` property that is higher than the current value of `tabIndex`. Once the Focus Manager reaches the highest `tabIndex` property, it returns to zero. So, in the following example, the `comment` object (probably a TextArea component) receives focus first, and then the `okButton` object receives focus:

```
comment.tabIndex = 1;
okButton.tabIndex = 2;
```

You can also use the Accessibility panel to assign a tab index value.

If nothing on the Stage has a tab index value, the Focus Manager uses the *depth* (stacking order, or *z*-order). The depth is set up primarily by the order in which components are dragged to the Stage; however, you can also use the Modify > Arrange > Bring to Front/Send to Back commands commands to determine the final depth.

To create a button that receives focus when a user presses Enter (Windows) or Return (Macintosh), set the `FocusManager.defaultPushButton` property to the instance name of the desired button, as shown here:

```
focusManager.defaultPushButton = okButton;
```

Note: The Focus Manager is sensitive to when objects are placed on the Stage (the depth order of objects) and not their relative positions on the Stage. This is different from the way Flash Player handles tabbing.

Using the Focus Manager to allow tabbing

You can use the Focus Manager to create a scheme that allows users to press the Tab key to cycle through objects in a Flash application. (Objects in the tab scheme are called *tab targets*.) The Focus Manager examines the `tabEnabled` and `tabChildren` properties of the objects' parents in order to locate the objects.

A movie clip can be either a container of tab targets, a tab target itself, or neither:

Movie clip type	tabEnabled	tabChildren
Container of tab targets	`false`	`true`
Tab target	`true`	`false`
Neither	`false`	`false`

Note: This is different from the default Flash Player behavior, in which a container's `tabChildren` property can be `undefined`.

Consider the following scenario. On the Stage of the main Timeline are two text fields (`txt1` and `txt2`) and a movie clip (`mc`) that contains a DataGrid component (`grid1`) and another text field (`txt3`). You would use the following code to allow users to press Tab and cycle through the objects in the following order: `txt1`, `txt2`, `grid1`, `txt3`.

Note: The FocusManager and TextField instances are enabled by default.

```
// let Focus Manager know mc has children;
// this overrides mc.focusEnabled=true;
mc.tabChildren=true;
mc.tabEnabled=false;
// set the tabbing sequence
txt1.tabIndex = 1;
txt2.tabIndex = 2;
mc.grid1.tabIndex = 3;
mc.txt3.tabIndex = 4;

// set initial focus to txt1
txt1.text = "focus";
focusManager.setFocus(txt1);
```

If your movie clip doesn't have an `onPress` or `onRelease` method or a `tabEnabled` property, it won't be seen by the Focus Manager unless you set `focusEnabled` to `true`. Input text fields are always in the tab scheme unless they are disabled.

If a Flash application is playing in a web browser, the application doesn't have focus until a user clicks somewhere in the application. Also, once a user clicks in the Flash application, pressing Tab can cause focus to jump outside the Flash application. To keep tabbing limited to objects inside the Flash application in Flash Player 7 ActiveX control, add the following parameter to the HTML `<object>` tag:

```
<param name="SeamlessTabbing" value="false"/>
```

Creating an application with the Focus Manager

The following procedure creates a focus scheme in a Flash application.

To create a focus scheme:

1. Drag the TextInput component from the Components panel to the Stage.
2. In the Property inspector, assign it the instance name **comment**.
3. Drag the Button component from the Components panel to the Stage.
4. In the Property inspector, assign it the instance name **okButton** and set the label parameter to **OK**.
5. In Frame 1 of the Actions panel, enter the following:

```
comment.tabIndex = 1;
okButton.tabIndex = 2;
focusManager.setFocus(comment);
function click(evt){
  trace(evt.type);
}
okButton.addEventListener("click", this);
```

This code sets the tab ordering. Although the comment field doesn't have a focus ring, it has initial focus, so you can start typing in the comment field without clicking on it.

Customizing the Focus Manager

You can change the color of the focus ring in the Halo theme by changing the value of the `themeColor` style, as in this example:

```
_global.style.setStyle("themeColor", "haloBlue");
```

The Focus Manager uses a FocusRect skin for drawing focus. This skin can be replaced or modified and subclasses can override `UIComponent.drawFocus` to draw custom focus indicators.

FocusManager class (API)

Inheritance MovieClip > UIObject class > UIComponent class > FocusManager

ActionScript Class Name mx.managers.FocusManager

You can use the Focus Manager to specify the order in which components receive focus when a user presses the Tab key to navigate in an application. You can also use the FocusManager class to set a button in your document that receives keyboard input when a user presses Enter (Windows) or Return (Macintosh).

Tip: In a class file that inherits from UIComponent, it is not good practice to refer to `_root.focusManager`. Every UIComponent instance inherits a `getFocusManager()` method, which returns a reference to the FocusManager instance responsible for controlling that component's focus scheme.

Method summary for the FocusManager class

The following table lists the methods of the FocusManager class.

Method	Description
`FocusManager.getFocus()`	Returns a reference to the object that has focus.
`FocusManager.sendDefaultPushButtonEvent()`	Sends a `click` event to listener objects registered to the default push button.
`FocusManager.setFocus()`	Sets focus to the specified object.

Methods inherited from the UIObject class

The following table lists the methods the FocusManager class inherits from the UIObject class.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the FocusManager class inherits from the UIComponent class.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Property summary for the FocusManager class

The following table lists the properties of the FocusManager class.

Property	Description
FocusManager.defaultPushButton	The object that receives a `click` event when a user presses the Return or Enter key.
FocusManager.defaultPushButtonEnabled	Indicates whether keyboard handling for the default push button is turned on (`true`) or off (`false`). The default value is `true`.
FocusManager.enabled	Indicates whether tab handling is turned on (`true`) or off (`false`). The default value is `true`.
FocusManager.nextTabIndex	The next value of the `tabIndex` property.

Properties inherited from the UIObject class

The following table lists the properties the FocusManager class inherits from the UIObject class.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the FocusManager class inherits from the UIComponent class.

Property	Description
`UIComponent.enabled`	Indicates whether the component can receive focus and input.
`UIComponent.tabIndex`	A number indicating the tab order for a component in a document.

Event summary for the FocusManager class

There are no events exclusive to the FocusManager class.

Events inherited from the UIObject class

The following table lists the events the FocusManager class inherits from the UIObject class.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the FocusManager class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

FocusManager.defaultPushButton

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
focusManager.defaultPushButton
```

Description

Property; specifies the default push button for an application. When the user presses Enter (Windows) or Return (Macintosh), the listeners of the default push button receive a `click` event. The default value is `undefined` and the data type of this property is object.

The Focus Manager uses the emphasized style declaration of the SimpleButton class to visually indicate the current default push button.

The value of the `defaultPushButton` property is always the button that has focus. Setting the `defaultPushButton` property does not give initial focus to the default push button. If there are several buttons in an application, the button that currently has focus receives the `click` event when Enter or Return is pressed. If some other component has focus when Enter or Return is pressed, the `defaultPushButton` property is reset to its original value.

Example

The following code sets the default push button to the `OKButton` instance:

```
focusManager.defaultPushButton = OKButton;
```

See also

`FocusManager.defaultPushButtonEnabled`,
`FocusManager.sendDefaultPushButtonEvent()`

FocusManager.defaultPushButtonEnabled

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
focusManager.defaultPushButtonEnabled
```

Description

Property; a Boolean value that determines if keyboard handling of the default push button is turned on (`true`) or not (`false`). Setting `defaultPushButtonEnabled` to `false` allows a component to receive the Return or Enter key and handle it internally. You must re-enable default push button handling by watching the component's `onKillFocus()` method (see `MovieClip.onKillFocus` in *Flash ActionScript Language Reference*) or `focusOut` event. The default value is `true`.

This property is for use by advanced component developers.

Example

The following code disables default push button handling:

```
focusManager.defaultPushButtonEnabled = false;
```

FocusManager.enabled

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
focusManager.enabled
```

Description

Property; a Boolean value that determines if tab handling is turned on (`true`) or not (`false`) for a particular group of focus objects. (For example, another pop-up window could have its own Focus Manager.) Setting `enabled` to `false` allows a component to receive the tab handling keys and handle them internally. You must re-enable the Focus Manager handling by watching the component's `onKillFocus()` method (see `MovieClip.onKillFocus` in *Flash ActionScript Language Reference*) or `focusOut` event. The default value is `true`.

Example

The following code disables tabbing:

```
focusManager.enabled = false;
```

FocusManager.getFocus()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
focusManager.getFocus()
```

Parameters

None.

Returns

A reference to the object that has focus.

Description

Method; returns a reference to the object that currently has focus.

Example

The following code sets the focus to `myOKButton` if the object that currently has focus is `myInputText`:

```
if (focusManager.getFocus() == myInputText)
{
  focusManager.setFocus(myOKButton);
}
```

See also

`FocusManager.setFocus()`

FocusManager.nextTabIndex

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
FocusManager.nextTabIndex
```

Description

Property; the next available tab index number. Use this property to dynamically set an object's `tabIndex` property.

Example

The following code gives the `mycheckbox` instance the next highest `tabIndex` value:

```
mycheckbox.tabIndex = focusManager.nextTabIndex;
```

See also

`UIComponent.tabIndex`

FocusManager.sendDefaultPushButtonEvent()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
focusManager.sendDefaultPushButtonEvent()
```

Parameters

None.

Returns

Nothing.

Description

Method; sends a `click` event to listener objects registered to the default push button. Use this method to programmatically send a `click` event.

Example

The following code triggers the default push button `click` event and fills in the user name and password fields when a user selects the CheckBox instance `chb` (the check box would be labeled "Automatic Login"):

```
name_txt.tabIndex = 1;
password_txt.tabIndex = 2;
chb.tabIndex = 3;
submit_ib.tabIndex = 4;

focusManager.defaultPushButton = submit_ib;

chbObj = new Object();
chbObj.click = function(o){
  if (chb.selected == true){
    name_txt.text = "Jody";
    password_txt.text = "foobar";
    focusManager.sendDefaultPushButtonEvent();
  } else {
    name_txt.text = "";
    password_txt.text = "";
  }
}
chb.addEventListener("click", chbObj);

submitObj = new Object();
submitObj.click = function(o){
  if (password_txt.text != "foobar"){
    trace("error on submit");
  } else {
    trace("Yeah! sendDefaultPushButtonEvent worked!");
  }
}
submit_ib.addEventListener("click", submitObj);
```

See also

`FocusManager.defaultPushButton`

FocusManager.setFocus()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
focusManager.setFocus(object)
```

Parameters

object A reference to the object to receive focus.

Returns

Nothing.

Description

Method; sets focus to the specified object. If the object to which you want to set focus is not on the main Timeline, use the following code:

```
_root.focusManager.setFocus(object);
```

Example

The following code sets focus to `myOKButton`:

```
focusManager.setFocus(myOKButton);
```

See also

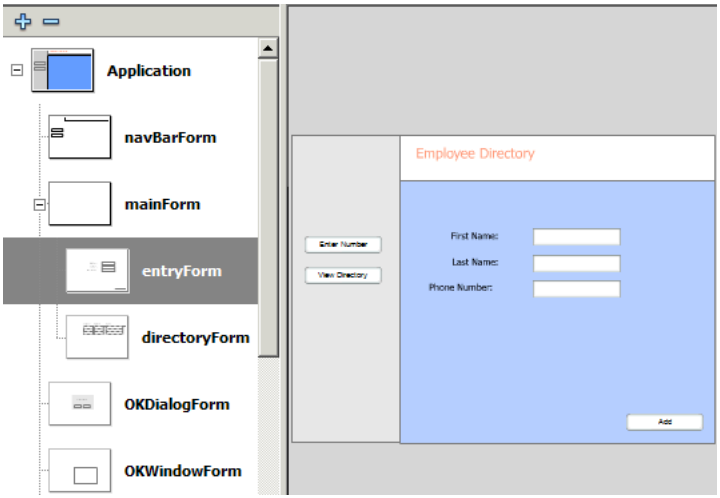
`FocusManager.getFocus()`

Form class (Flash Professional only)

The Form class provides the runtime behavior of forms you create in the Screen Outline pane in Flash MX Professional 2004. For an overview of working with screens, see Chapter 12, "Working with Screens (Flash Professional Only)," in *Using Flash*.

Using the Form class (Flash Professional only)

Forms function as containers for graphic objects—user interface elements in an application, for example—as well as application states. You can use the Screen Outline pane to visualize the different states of an application that you're creating, where each form is a different application state. For example, the following illustration shows the Screen Outline pane for an example application designed using forms.



Screen Outline view of sample form application

This illustration shows the outline for a sample application called Employee Directory, which consists of several forms. The form named entryForm (selected in the above illustration) contains several user interface objects, including input text fields, labels, and a push button. The developer can easily present this form to the user by toggling its visibility (using the `Form.visible` property), while simultaneously toggling the visibility of other forms, as well.

Using the Behaviors panel you can also attach behaviors and controls to forms. For more information about adding transitions and controls to screens, see "Creating controls and transitions for screens with behaviors (Flash Professional only)" in *Using Flash*.

Because the Form class extends the Loader class, you can easily load external content (a SWF or JPEG file) into a form. For example, the contents of a form could be a separate SWF file, which itself might contain forms. In this way, you can make your form applications modular, which makes maintaining the applications easier, and also reduces initial download time. For more information, see "Loading external content into screens (Flash Professional only)" on page 651.

Form parameters

You can set the following authoring parameters for each Form instance in the Property inspector or in the Component inspector:

autoload indicates whether the content specified by the contentPath parameter should load automatically (`true`), or wait to load until the `Loader.load()` method is called (`false`). The default value is `true`.

contentPath specifies the contents of the form. This can be the linkage identifier of a movie clip or an absolute or relative URL for a SWF or JPEG file to load into the slide. By default, loaded content is clipped to fit the slide.

visible specifies whether the form is visible (`true`) or not (`false`) when it first loads.

Form class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > Loader component > Screen class (Flash Professional only) > Form

ActionScript Class Name mx.screens.Form

The Form class provides the runtime behavior of forms you create in the Screen Outline pane in Flash MX Professional 2004.

Method summary for the Form class

The following table lists methods of the Form class.

Method	Description
`Form.getChildForm()`	Returns the child form at a specified index.

Methods inherited from the UIObject class

The following table lists the methods the Form class inherits from the UIObject class. When calling these methods from the Form object, use the syntax *`formInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.

Method	Description
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Form class inherits from the UIComponent class. When calling these methods from the Form object, use the syntax *formInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Methods inherited from the Loader class

The following table lists the methods the Form class inherits from the Loader class. When calling these methods from the Form object, use the syntax *formInstance.methodName*.

Method	Description
Loader.load()	Loads the content specified by the contentPath property.

Methods inherited from the Screen class

The following table lists the methods the Form class inherits from the Screen class. When calling these methods from the Form object, use the syntax *formInstance.methodName*.

Method	Description
Screen.getChildScreen()	Returns the child screen of this screen at a particular index.

Property summary for the Form class

The following table lists the properties that are exclusive to the Form class.

Property	Description
Form.currentFocusedForm	Read-only; returns the form that contains the global current focus.
Form.indexInParentForm	Read-only; returns the index (zero-based) of this form in its parent's list of subforms.
Form.numChildForms	Read-only; returns the number of child forms that this form contains.
Form.parentIsForm	Read-only; specifies whether the parent object of this form is also a form.
Form.parentForm	Read-only; reference to the form's parent form.

Property	Description
Form.parentForm	Read-only; returns the root of the form tree, or subtree, that contains the form.
Form.visible	Specifies whether the form is visible when its parent form, slide, movie clip, or SWF file is visible.

Properties inherited from the UIObject class

The following table lists the properties the Form class inherits from the UIObject class. When accessing these properties from the Form object, use the syntax *formInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Form class inherits from the UIComponent class. When accessing these properties from the Form object, use the syntax *formInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the Loader class

The following table lists the properties the Form class inherits from the Loader class. When accessing these properties from the Form object, use the syntax *formInstance*.*propertyName*.

Property	Description
Loader.autoLoad	A Boolean value that indicates whether the content loads automatically (true) or you must call Loader.load() (false).
Loader.bytesLoaded	A read-only property that indicates the number of bytes that have been loaded.
Loader.bytesTotal	A read-only property that indicates the total number of bytes in the content.
Loader.content	A reference to the content of the loader. This property is read-only.
Loader.contentPath	A string that indicates the URL of the content to be loaded.
Loader.percentLoaded	A number that indicates the percentage of loaded content. This property is read-only.
Loader.scaleContent	A Boolean value that indicates whether the content scales to fit the loader (true), or the loader scales to fit the content (false).

Properties inherited from the Screen class

The following table lists the properties the Form class inherits from the Screen class. When accessing these properties from the Form object, use the syntax *formInstance*.*propertyName*.

Property	Description
Screen.currentFocusedScreen	Read-only; returns the screen that contains the global current focus.
Screen.indexInParent	Read-only; returns the screen's index (zero-based) in its parent screen's list of child screens.
Screen.numChildScreens	Read-only; returns the number of child screens contained by the screen.
Screen.parentIsScreen	Read-only; returns a Boolean (true or false) value that indicates whether the screen's parent object is itself a screen.
Screen.rootScreen	Read-only; returns the root screen of the tree or subtree that contains the screen.

Event summary for the Form class

There are no events exclusive to the Form class.

Events inherited from the UIObject class

The following table lists the events the Form class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Form class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Events inherited from the Loader class

The following table lists the events the Form class inherits from the Loader class.

Event	Description
Loader.complete	Triggered when the content finished loading.
Loader.progress	Triggered while content is loading.

Events inherited from the Screen class

The following table lists the events the Form class inherits from the Screen class.

Event	Description
Screen.allTransitionsInDone	Broadcast when all "in" transitions applied to a screen have finished.
Screen.allTransitionsOutDone	Broadcast when all "out" transitions applied to a screen have finished.
Screen.mouseDown	Broadcast when the mouse button was pressed over an object (shape or movie clip) directly owned by the screen.
Screen.mouseDownSomewhere	Broadcast when the mouse button was pressed somewhere on the Stage, but not necessarily on an object owned by this screen.

Event	Description
Screen.mouseMove	Broadcast when the mouse is moved while over a screen.
Screen.mouseOut	Broadcast when the mouse is moved from inside the screen to outside it.
Screen.mouseOver	Broadcast when the mouse is moved from outside this screen to inside it.
Screen.mouseUp	Broadcast when the mouse button was released over an object (shape or movie clip) directly owned by the screen.
Screen.mouseUpSomewhere	Broadcast when the mouse button was released somewhere on the Stage, but not necessarily over an object owned by this screen.

Form.currentFocusedForm

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
mx.screens.Form.currentFocusedForm
```

Description

Property (read-only); returns the Form object that contains the global current focus. The actual focus may be on the form itself, or on a movie clip, text object, or component inside that form. May be `null` if there is no current focus.

Example

The following code, attached to a button (not shown), displays the name of the form with the current focus.

```
trace("The form with the current focus is: " +
  mx.screens.Form.currentFocusedForm);
```

Form.getChildForm()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myForm.getChildForm(childIndex)
```

Parameters

childIndex A number that indicates the zero-based index of the child form to return.

Returns

A Form object.

Description

Method; returns the child form of *myForm* whose index is *childIndex*.

Example

The following example displays in the Output panel the names of all the child Form objects belonging to the root Form object named `application`.

```
for (var i:Number = 0; i < _root.application.numChildForms; i++) {
    var childForm:mx.screens.Form = _root.application.getChildForm(i);
    trace(childForm._name);
}
```

See also

`Form.numChildForms`

Form.indexInParentForm

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myForm`.indexInParentForm`

Description

Property (read-only); contains the zero-based index of *myForm* in its parent's list of child forms. If the parent object of *myForm* is a screen but not a form (for example, if it is a slide), `indexInParentForm` is always 0.

Example

```
var myIndex:Number = myForm.indexInParent;
if (myForm == myForm._parent.getChildForm(myIndex)) {
  trace("I'm where I should be");
}
```

See also

`Form.getChildForm()`

Form.numChildForms

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myForm.numChildForms

Description

Property (read-only); the number of child forms contained by *myForm* that are derived directly from the class mx.screens.Form. This property does not include any slides that are contained by *myForm*; it contains only forms.

Note: When using a custom ActionScript 2.0 class that extends mx.screens.Form, the form isn't counted in the numChildForms property.

Example

The following code iterates over all the child forms contained in myForm and displays their names in the Output panel.

```
var howManyKids:Number = myForm.numChildForms;
for(i=0; i<howManyKids; i++) {
  var childForm = myForm.getChildForm(i);
  trace(childForm._name);
}
```

See also

Form.getChildForm()

Form.parentIsForm

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myForm.parentIsForm

Description

Property (read-only): returns a Boolean value indicating whether the specified form's parent object is also a form (true) or not (false). If this property is false, *myForm* is at the root of its form hierarchy.

Example

```
if (myForm.parentIsForm) {
  trace("I have "+myForm._parent.numChildScreens+" sibling screens");
} else {
  trace("I am the root form and have no siblings");
}
```

Form.parentForm

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myForm.parentForm
```

Description

Property (read-only): a reference to the form's parent form.

Example

The following example code resides on a screen named `myForm` that is a child of the default `form1` screen created when you choose Flash Form Application from the New Document dialog box.

```
onClipEvent(keyDown){
  var parentForm:mx.screens.Form = this.parentForm;
  trace(parentForm);
}
// output: _level0.application.form1
```

Form.rootForm

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myForm.rootForm
```

Description

Property (read-only); returns the form at the top of the form hierarchy that contains *myForm*. If *myForm* is contained by an object that is not a form (that is, a slide), this property returns *myForm*.

Example

In the following example, a reference to the root form of `myForm` is placed in a variable named `root`. If the value assigned to `root` refers to `myForm`, then `myForm` is at the top of its form tree.

```
var root:mx.screens.Form = myForm.rootForm;
if(rootForm == myForm) {
  trace("myForm is the top form in its tree");
}
```

Form.visible

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myForm`.visible`

Description

Property; determines whether *myForm* is visible when its parent form, slide, movie clip, or SWF file is visible. You can also set this property using the Property inspector in the Flash authoring environment.

When this property is set to `true`, *myForm* receives a `reveal` event; when set to `false`, *myForm* receives a `hide` event. You can attach transitions to forms that execute when a form receives one of these events. For more information on adding transitions to screens, see "Creating controls and transitions for screens with behaviors (Flash Professional only)" in *Using Flash*.

Example

The following code, on a Timeline frame, sets the `visible` property of the form that contains the button to `false`.

```
btnOk.addEventListener("click", btnOkClick);
function btnOkClick(eventObj:Object):Void {
  eventObj.target._parent.visible = false;
}
```

Iterator interface (Flash Professional only)

ActionScript Class Name mx.utils.Iterator

The Iterator interface lets you step through the objects that a collection contains.

Method summary for the Iterator interface

The following table lists the methods of the Iterator interface.

Method	Description
`Iterator.hasNext()`	Indicates whether the iterator has more items.
`Iterator.next()`	Returns the next item in the iteration.

Iterator.hasNext()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

iterator.hasNext()

Returns

A Boolean value that indicates whether there are (`true`) or are not (`false`) more instances in the iterator.

Description

Method; indicates whether there are more instances in the iterator. You typically use this method in a `while` statement when looping through an iterator.

Example

The following example uses the `hasNext()` method to control looping through the iterator of items in a collection:

```
on (click) {
  var myColl:mx.utils.Collection;
  myColl = _parent.thisShelf.MyCompactDisks;
  var itr:mx.utils.Iterator = myColl.getIterator();
  while (itr.hasNext()) {
    var cd:CompactDisk = CompactDisk(itr.next());
    var title:String = cd.Title;
    var artist:String = cd.Artist;
    trace("Title: "+title+" Artist: "+artist);
  }
}
```

Iterator.next()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
iterator.next()
```

Returns

An object that is the next item in the iterator.

Description

Method; returns an instance of the next item in the iterator. You must cast this instance to the correct type.

Example

The following example uses the `next()` method to access the next item in a collection:

```
on (click) {
  var myColl:mx.utils.Collection;
  myColl = _parent.thisShelf.MyCompactDisks;
  var itr:mx.utils.Iterator = myColl.getIterator();
  while (itr.hasNext()) {
    var cd:CompactDisk = CompactDisk(itr.next());
    var title:String = cd.Title;
    var artist:String = cd.Artist;
    trace("Title: "+title+" Artist: "+artist);
  }
}
```

Label component

A label component is a single line of text. You can specify that a label be formatted with HTML. You can also control the alignment and size of a label. Label components don't have borders, cannot be focused, and don't broadcast any events.

A live preview of each Label instance reflects changes made to parameters in the Property inspector or Component inspector during authoring. The label doesn't have a border, so the only way to see its live preview is to set its text parameter. The autoSize parameter is not supported in live preview.

When you add the Label component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code to enable accessibility:

```
mx.accessibility.LabelAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the label component

Use a Label component to create a text label for another component in a form, such as a "Name:" label to the left of a TextInput field that accepts a user's name. If you're building an application using components based on version 2 of the Macromedia Component Architecture, it's a good idea to use a Label component instead of a plain text field because you can use styles to maintain a consistent look and feel.

If you want to rotate a Label component, you must embed the fonts. See "Using styles with the Label component" on page 444.

Label parameters

You can set the following authoring parameters for each Label component instance in the Property inspector or in the Component inspector:

text indicates the text of the label; the default value is `Label`.

html indicates whether the label is formatted with HTML (`true`) or not (`false`). If this parameter is set to `true`, a label cannot be formatted with styles. The default value is `false`.

autoSize indicates how the label sizes and aligns to fit the text. The default value is `none`. The parameter can have any of the following four values:

- `none`, which specifies that the label doesn't resize or align to fit the text.
- `left`, which specifies that the right and bottom sides of the label resize to fit the text. The left and top sides don't resize.
- `center`, which specifies that the bottom side of the label resizes to fit the text. The horizontal center of the label stays anchored at its original horizontal center position.
- `right`, which specifies that the left and bottom sides of the label resize to fit the text. The top and right side don't resize.

Note: The Label component's `autoSize` property is different from the built-in ActionScript TextField object's `autoSize` property.

You can write ActionScript to set additional options for Label instances using its methods, properties, and events. For more information, see "Label class" on page 445.

Creating an application with the Label component

The following procedure explains how to add a Label component to an application while authoring. In this example, the label is beside a combo box with dates in a shopping cart application.

To create an application with the Label component:

1. Drag a Label component from the Components panel to the Stage.
2. In the Component inspector, enter **Expiration Date** for the label parameter.

Customizing the Label component

You can transform a Label component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. You can also set the autoSize authoring parameter; setting this parameter doesn't change the bounding box in the live preview, but the label does resize. For more information, see "Label parameters" on page 443. At runtime, use the `setSize()` method (see `UIObject.setSize()`) or `Label.autoSize`.

Using styles with the Label component

You can set style properties to change the appearance of a label instance. All text in a Label component instance must share the same style. For example, you can't set the `color` style to `"blue"` for one word in a label and to `"red"` for the second word in the same label.

If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information about styles, see "Using styles to customize component color and text" on page 67.

A Label component supports the following styles:

Style	Theme	Description
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.

Style	Theme	Description
fontSize	Both	The point size for the font. The default value is 10.
fontStyle	Both	The font style: either "normal" or "italic". The default value is "normal".
fontWeight	Both	The font weight: either "none" or "bold". The default value is "none". All components can also accept the value "normal" in place of "none" during a setStyle() call, but subsequent calls to getStyle() will return "none".
textAlign	Both	The text alignment: either "left", "right", or "center". The default value is "left".
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".

Using skins with the Label component

The Label component does not have any visual elements to skin.

Label class

Inheritance MovieClip > UIObject class > Label

ActionScript Class Name mx.controls.Label

The properties of the Label class allow you at runtime to specify text for the label, indicate whether the text can be formatted with HTML, and indicate whether the label auto-sizes to fit the text.

Setting a property of the Label class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

Each component class has a version property, which is a class property. Class properties are available only on the class itself. The version property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.Label.version);
```

Note: The code trace(myLabelInstance.version); returns undefined.

Method summary for the Label class

There are no methods exclusive to the Label class.

Methods inherited from the UIObject class

The following table lists the methods the Label class inherits from the UIObject class. When calling these methods from the Label object, use the form *labelInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.

Method	Description
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Property summary for the Label class

The following table lists properties of the Label class.

Property	Description
Label.autoSize	A string that indicates how a label sizes and aligns to fit the value of its `text` property. There are four possible values: `"none"`, `"left"`, `"center"`, and `"right"`. The default value is `"none"`.
Label.html	A Boolean value that indicates whether a label can be formatted with HTML (`true`) or not (`false`).
Label.text	The text on the label.

Properties inherited from the UIObject class

The following table lists the properties the Label class inherits from the UIObject class. When accessing these properties, use the form *`labelInstance.propertyName`*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.

Property	Description
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Event summary for the Label class

There are no events exclusive to the Label class.

Events inherited from the UIObject class

The following table lists the events the Label class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Label.autoSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

labelInstance.autoSize

Description

Property; a string that indicates how a label sizes and aligns to fit the value of its text property. There are four possible values: "none", "left", "center", and "right". The default value is "none".

- none The label doesn't resize or align to fit the text.
- left The right and bottom sides of the label resize to fit the text. The left and top sides don't resize.

- `center` The bottom side of the label resizes to fit the text. The horizontal center of the label stays anchored at its original horizontal center position.
- `right` The left and bottom sides of the label resize to fit the text. The top and right sides don't resize.

Note: The Label component's `autoSize` property is different from the built-in ActionScript TextField object's `autoSize` property.

Label.html

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`labelInstance``.html`

Description

Property; a Boolean value that indicates whether the label can be formatted with HTML (`true`) or not (`false`). The default value is `false`. Label components with the `html` property set to `true` cannot be formatted with styles.

To retrieve plain text from HTML-formatted text, set the `HTML` property to `false` and then access the `text` property. This will remove the HTML formatting, so you may want to copy the label text to an offscreen Label or TextArea component before you retrieve the plain text.

Example

The following example sets the `html` property to `true` so the label can be formatted with HTML. The `text` property is then set to a string that includes HTML formatting.

```
lbl.html = true;
lbl.text = "The <b>Royal</b> Nonesuch";
```

The word "Royal" displays in bold.

Label.text

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`labelInstance``.text`

Description

Property; the text of a label. The default value is `"Label"`.

Example

The following code sets the `text` property of the Label instance `labelControl` and sends the value to the Output panel:

```
labelControl.text = "The Royal Nonesuch";
trace(labelControl.text);
```

List component

The List component is a scrollable single- or multiple-selection list box. A list can also display graphics, including other components. You add the items displayed in the list by using the Values dialog box that appears when you click in the labels or data parameter fields. You can also use the `List.addItem()` and `List.addItemAt()` methods to add items to the list.

The List component uses a zero-based index, where the item with index 0 is the top item displayed. When adding, removing, or replacing list items using the List class methods and properties, you may need to specify the index of the list item.

The list receives focus when you click it or tab to it, and you can then use the following keys to control it:

Key	Description
Alphanumeric keys	Jump to the next item that has `Key.getAscii()` as the first character in its label.
Control	Toggle key that allows multiple noncontiguous selections and deselections.
Down Arrow	Selection moves down one item.
Home	Selection moves to the top of the list.
Page Down	Selection moves down one page.
Page Up	Selection moves up one page.
Shift	Allows for contiguous selection.
Up Arrow	Selection moves 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 ten-line drop-down list will show items 0-9, 9-18, 18-27, and so on, with one item overlapping per page.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each List instance on the Stage reflects changes made to parameters in the Property inspector or Component inspector during authoring.

When you add the List component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code to enable accessibility:

```
mx.accessibility.ListAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the List component

You can set up a list so that users can make either single or multiple selections. For example, a user visiting an e-commerce website needs to select which item to buy. There are 30 items, and the user scrolls through a list and selects one by clicking it.

You can also design a list that uses custom movie clips as rows so you can display more information to the user. For example, in an e-mail application, each mailbox could be a List component and each row could have icons to indicate priority and status.

Understanding the design of the List component

When you design an application with the List component, or any component that extends the List class, it is helpful to understand how the list was designed. The following are some fundamental assumptions and requirements that Macromedia used when developing the List class:

- Keep it small, fast, and simple.

  Don't make something more complicated than absolutely necessary. This was the prime design directive. Most of the requirements listed below are based on this directive.

- Lists have uniform row heights.

  Every row must be the same height; the height can be set during authoring or at runtime.

- Lists must scale to thousands of records.

- Lists don't measure text.

  This restriction has the most potential ramifications. Because a list must scale to thousands of records, any one of which could contain an unusually long string, it shouldn't grow to fit the largest string of text within it, or add a horizontal scroll bar in "auto" mode. Also, measuring thousands of strings would be too intensive. The compromise is the `maxHPosition` property, which, when `vScrollPolicy` is set to `"on"`, gives the list extra buffer space for scrolling.

  If you know you're likely to deal with long strings, turn `hScrollPolicy` to `"on"`, and add a 200-pixel `maxHPosition` value to your List or Tree component. A user is more or less guaranteed to be able to scroll to see everything. The DataGrid component, however, does support `"auto"` as an `hScrollPolicy` value, because it measures columns (which are the same width per item), not text.

  The fact that lists don't measure text also explains why lists have uniform row heights. Sizing individual rows to fit text would require intensive measuring. For example, if you wanted to accurately show the scroll bars on a list with nonuniform row height, you'd need to premeasure every row.

- Lists perform worse as a function of their visible rows.

  Although lists can display 5000 records, they can't render 5000 records at once. The more visible rows (specified by the `rowCount` property) you have on the Stage, the more work the list must to do to render. Limiting the number of visible rows, if at all possible, is the best solution.

- Lists aren't tables.

  For example, DataGrid components, which extend the List class, are intended to provide an interface for many records. They're not designed to display complete information; they're designed to display enough information so that users can drill down to see more. The message view in Microsoft Outlook is a prime example. You don't read the entire e-mail in the grid; the mail would be difficult to read and the client would perform terribly. Outlook displays enough information so that a user can drill into the post to see the details.

List parameters

You can set the following authoring parameters for each List component instance in the Property inspector or in the Component inspector:

data is an array of values that populate the data of the list. The default value is `[]` (an empty array). There is no equivalent runtime property.

labels is an array of text values that populate the label values of the list. The default value is `[]` (an empty array). There is no equivalent runtime property.

multipleSelection is a Boolean value that indicates whether you can select multiple values (`true`) or not (`false`). The default value is `false`.

rowHeight indicates the height, in pixels, of each row. The default value is 20. Setting a font does not change the height of a row.

You can write ActionScript to set additional options for List instances using its methods, properties, and events. For more information, see "List class" on page 456.

Creating an application with the List component

The following procedure explains how to add a List component to an application while authoring. In this example, the list is a sample with three items.

To add a simple List component to an application:

1. Drag a List component from the Components panel to the Stage.
2. Select the list and select Modify > Transform to resize it to fit your application.
3. In the Property inspector, do the following:
   - Enter the instance name **myList**.
   - Enter **Item1**, **Item2**, and **Item3** for the labels parameter.
   - Enter **item1.html**, **item2.html**, **item3.html** for the data parameter.
4. Select Control > Test Movie to see the list with its items.

   You could use the data property values in your application to open HTML files.

To populate a List instance with a data provider:

1. Drag a List component from the Components panel to the Stage.
2. Select the list and select Modify > Transform to resize it to fit your application.
3. In the Actions panel, enter the instance name **myList**.

4. Select Frame 1 of the Timeline and, in the Actions panel, enter the following:

```
myList.dataProvider = myDP;
```

If you have defined a data provider named `myDP`, the list will fill with data. (For more information about data providers, see `List.dataProvider`.)

5. Select Control > Test Movie to see the list with its items.

Customizing the List component

You can transform a List component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `List.setSize()` method (see `UIObject.setSize()`).

When a list is resized, the rows of the list shrink horizontally, clipping any text within them. Vertically, the list adds or removes rows as needed. Scroll bars position themselves automatically. For more information about scroll bars, see "ScrollPane component" on page 668.

Using styles with the List component

You can set style properties to change the appearance of a List component.

A List component uses the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`alternatingRowColors`	Both	Specifies colors for rows in an alternating pattern. The value can be an array of two or more colors, for example, 0xFF00FF, 0xCC6699, and 0x996699. Unlike single-value color styles, `alternatingRowColors` does not accept color names; the values must be numeric color codes. By default, this style is not set and `backgroundColor` is used in its place for all rows.
`backgroundColor`	Both	The background color of the list. The default color is white and is defined on the class style declaration. This style is ignored if `alternatingRowColors` is specified.
`backgroundDisabledColor`	Both	The background color when the component's `enabled` property is set to `"false"`. The default value is 0xDDDDDD (medium gray).
`border styles`	Both	The List component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style is `"inset"`.
`color`	Both	The text color.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).

Style	Theme	Description
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textAlign`	Both	The text alignment: either `"left"`, `"right"`, or `"center"`. The default value is `"left"`.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.
`textIndent`	Both	A number indicating the text indent. The default value is 0.
`defaultIcon`	Both	The name of the default icon to display on each row. The default value is `undefined`, which means no icon is displayed.
`repeatDelay`	Both	The number of milliseconds of delay between when a user first presses a button in the scrollbar and when the action begins to repeat. The default value is 500, half a second.
`repeatInterval`	Both	The number of milliseconds between automatic clicks when a user holds the mouse button down on a button in the scrollbar. The default value is 35.
`rollOverColor`	Both	The background color of a rolled-over row. The default value is 0xE3FFD6 (bright green) with the Halo theme and 0xAAAAAA (light gray) with the Sample theme. When `themeColor` is changed through a `setStyle()` call, the framework sets `rollOverColor` to a value related to the `themeColor` chosen.
`selectionColor`	Both	The background color of a selected row. The default value is a 0xCDFFC1 (light green) with the Halo theme and 0xEEEEEE (very light gray) with the Sample theme. When `themeColor` is changed through a `setStyle()` call, the framework sets `selectionColor` to a value related to the `themeColor` chosen.
`selectionDuration`	Both	The length of the transition from a normal to selected state or back from selected to normal, in milliseconds. The default value is 200.

Style	Theme	Description
selectionDisabledColor	Both	The background color of a selected row. The default value is a 0xDDDDDD (medium gray). Because the default value for this property is the same as the default for backgroundDisabledColor, the selection is not visible when the component is disabled unless one of these style properties is changed.
selectionEasing	Both	A reference to the easing equation used to control the transition between selection states. This applies only for the transition from a normal to a selected state. The default equation uses a sine in/out formula. For more information, see "Customizing component animations" on page 75.
textRollOverColor	Both	The color of text when the mouse pointer rolls over it. The default value is 0x2B333C (dark gray). This style is important when you set rollOverColor, because the two settings must complement each other so that text is easily viewable during a rollover.
textSelectedColor	Both	The color of text in the selected row. The default value is 0x005F33 (dark gray). This style is important when you set selectionColor, because the two settings must complement each other so that text is easily viewable while selected.
useRollOver	Both	Determines whether rolling over a row activates highlighting. The default value is true.

Setting styles for all List components in a document

The List class inherits from the ScrollSelectList class. The default class-level style properties are defined on the ScrollSelectList class, which the Menu component and all List-based components extend. You can set new default style values on this class directly, and the new settings will be reflected in all affected components.

```
_global.styles.ScrollSelectList.setStyle("backgroundColor", 0xFF00AA);
```

To set a style property on the List and List-based components only, you can create a new CSSStyleDeclaration instance and store it in _global.styles.List.

```
import mx.styles.CSSStyleDeclaration;
if (_global.styles.List == undefined) {
  _global.styles.List = new CSSStyleDeclaration();
}
_global.styles.List.setStyle("backgroundColor", 0xFF00AA);
```

When creating a new class-level style declaration, you will lose all default values provided by the ScrollSelectList declaration. This includes backgroundColor, which is required for supporting mouse events. To create a class-level style declaration and preserve defaults, use a for..in loop to copy the old settings to the new declaration.

```
var source = _global.styles.ScrollSelectList;
var target = _global.styles.List;
for (var style in source) {
```

```
  target.setStyle(style, source.getStyle(style));
}
```

To provide styles for the List component but not for components that extend List (DataGrid and Tree), you must provide class-level style declarations for these subclasses.

```
import mx.styles.CSSStyleDeclaration;
if (_global.styles.DataGrid == undefined) {
  _global.styles.DataGrid = new CSSStyleDeclaration();
}
_global.styles.DataGrid.setStyle("backgroundColor", 0xFFFFFF);
if (_global.styles.Tree == undefined) {
  _global.styles.Tree = new CSSStyleDeclaration();
}
_global.styles.Tree.setStyle("backgroundColor", 0xFFFFFF);
```

For more information about class-level styles, see "Setting styles for a component class" on page 71.

Using skins with the List component

The List component uses an instance of RectBorder for its border and scroll bars for scrolling images. For more information about skinning these visual elements, see "RectBorder class" on page 647 and "Using skins with the UIScrollBar component" on page 831.

List class

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > ScrollSelectList > List

ActionScript Class Name mx.controls.List

The List component is composed of three parts: items, rows, and a data provider.

An *item* is an ActionScript object used for storing the units of information in the list. A list can be thought of as an array; each indexed space of the array is an item. An item is an object that typically has a `label` property that is displayed and a `data` property that is used for storing data.

A *row* is a component that is used to display an item. Rows are either supplied by default by the list (the SelectableRow class is used), or you can supply them, usually as a subclass of the SelectableRow class. The SelectableRow class implements the CellRenderer API, which is the set of properties and methods that allow the list to manipulate each row and send data and state information (for example, size, selected, and so on) to the row for display.

A data provider is a data model of the list of items in a list. Any array in the same frame as a list is automatically given methods that let you manipulate data and broadcast changes to multiple views. You can build an Array instance or get one from a server and use it as a data model for multiple lists, combo boxes, data grids, and so on. The List component has methods that proxy to its data provider (for example, `addItem()` and `removeItem()`). If no external data provider is provided to the list, these methods create a data provider instance automatically, which is exposed through `List.dataProvider`.

To add a List component to the tab order of an application, set its `tabIndex` property (see `UIComponent.tabIndex`). The List component uses the Focus Manager to override the default Flash Player focus rectangle and draw a custom focus rectangle with rounded corners. For more information, see "Creating custom focus navigation" on page 50.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.List.version);
```

Note: The code `trace(myListInstance.version);` returns `undefined`.

Method summary for the List class

The following table lists methods of the List class.

Method	Description
`List.addItem()`	Adds an item to the end of the list.
`List.addItemAt()`	Adds an item to the list at the specified index.
`List.getItemAt()`	Returns the item at the specified index.
`List.removeAll()`	Removes all items from the list.
`List.removeItemAt()`	Removes the item at the specified index.
`List.replaceItemAt()`	Replaces the item at the specified index with another item.
`List.setPropertiesAt()`	Applies the specified properties to the specified item.
`List.sortItems()`	Sorts the items in the list according to the specified compare function.
`List.sortItemsBy()`	Sorts the items in the list according to a specified property.

Methods inherited from the UIObject class

The following table lists the methods the List class inherits from the UIObject class. When calling these methods, use the form *`listInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.

Method	Description
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the List class inherits from the UIComponent class. When calling these methods, use the form *listInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the List class

The following table lists properties of the List class.

Property	Description
List.cellRenderer	Assigns the class or symbol to use to display each row of the list.
List.dataProvider	The source of the list items.
List.hPosition	The horizontal position of the list.
List.hScrollPolicy	Indicates whether the horizontal scroll bar is displayed ("on") or not ("off").
List.iconField	A field in each item to be used to specify icons.
List.iconFunction	A function that determines which icon to use.
List.labelField	Specifies a field of each item to be used as label text.
List.labelFunction	A function that determines which fields of each item to use for the label text.
List.length	The number of items in the list. This property is read-only.
List.maxHPosition	The number of pixels the list can scroll to the right, when List.hScrollPolicy is set to "on".
List.multipleSelection	Indicates whether multiple selection is allowed in the list (true) or not (false).
List.rowCount	The number of rows that are at least partially visible in the list.
List.rowHeight	The pixel height of every row in the list.
List.selectable	Indicates whether the list is selectable (true) or not (false).
List.selectedIndex	The index of a selection in a single-selection list.
List.selectedIndices	An array of the selected items in a multiple-selection list.

Property	Description
List.selectedItem	The selected item in a single-selection list. This property is read-only.
List.selectedItems	The selected item objects in a multiple-selection list. This property is read-only.
List.vPosition	The topmost visible item of the list.
List.vScrollPolicy	Indicates whether the vertical scroll bar is displayed ("on"), not displayed ("off"), or displayed when needed ("auto").

Properties inherited from the UIObject class

The following table lists the properties the List class inherits from the UIObject class. When accessing these properties, use the form *listInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the List class inherits from the UIComponent class. When accessing these properties, use the form *listInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the List class

The following table lists events that of the List class.

Event	Description
`List.change`	Broadcast whenever user interaction causes the selection to change.
`List.itemRollOut`	Broadcast when the pointer rolls over and then off of list items.
`List.itemRollOver`	Broadcast when the pointer rolls over list items.
`List.scroll`	Broadcast when a list is scrolled.

Events inherited from the UIObject class

The following table lists the events the List class inherits from the UIObject class.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the List class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

List.addItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.addItem(label[, data])
listInstance.addItem(itemObject)
```

Parameters

label A string that indicates the label for the new item.

data The data for the item. This parameter is optional and can be of any data type.

itemObject An item object that usually has `label` and `data` properties.

Returns

The index at which the item was added.

Description

Method; adds a new item to the end of the list.

In the first usage example, an item object is always created with the specified `label` property, and, if specified, the `data` property.

The second usage example adds the specified item object.

Calling this method modifies the data provider of the List component. If the data provider is shared with other components, those components will update as well.

Example

Both of the following lines of code add an item to the `myList` instance. To try this code, drag a List component to the Stage and give it the instance name **myList**. Add the following code to Frame 1 in the Timeline:

```
myList.addItem("this is an Item");
myList.addItem({label:"Gordon",age:"very old",data:123});
```

List.addItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.addItemAt(index, label[, data])
listInstance.addItemAt(index, itemObject)
```

Parameters

index A number greater than or equal to 0 that indicates the position of the item.

label A string that indicates the label for the new item.

data The data for the item. This parameter is optional and can be of any data type.

itemObject An item object that usually has `label` and `data` properties.

Returns

The index at which the item was added.

Description

Method; adds a new item to the position specified by the *index* parameter.

In the first usage example, an item object is always created with the specified `label` property, and, if specified, the `data` property.

The second usage example adds the specified item object.

Calling this method modifies the data provider of the List component. If the data provider is shared with other components, those components will update as well.

Example

The following line of code adds an item to the third index position, which is the fourth item in the list:

```
myList.addItemAt(3,{label:'Red',data:0xFF0000});
```

List.cellRenderer

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance`.cellRenderer`

Description

Property; assigns the cell renderer to use for each row of the list. This property must be a class object reference or a symbol linkage identifier. Any class used for this property must implement the CellRenderer API.

Example

The following example uses a linkage identifier to set a new cell renderer:

```
myList.cellRenderer = "ComboBoxCell";
```

List.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(change){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  // your code here
}
listInstance.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners when the selected index of the list changes as a result of user interaction.

The first usage example uses an `on()` handler and must be attached directly to a List instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the List instance `myBox`, sends "_level0.myBox" to the Output panel:

```
on(click){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`listInstance`*) dispatches an event (in this case, `change`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class (API)" on page 415.

Finally, you call the `addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

Example

The following example sends the instance name of the component that generated the `change` event to the Output panel:

```
form.change = function(eventObj){
  trace("Value changed to " + eventObj.target.value);
}
myList.addEventListener("change", form);
```

See also

`EventDispatcher.addEventListener()`

List.dataProvider

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listinstance.dataProvider

Description

Property; the data model for items viewed in a list. The value of this property can be an array or any object that implements the DataProvider API. The default value is `[]`. For more information, see "DataProvider API" on page 290.

The List component, like other data-aware components, adds methods to the Array object's prototype so that they conform to the DataProvider API. Therefore, any array that exists at the same time as a list automatically has all the methods (`addItem()`, `getItemAt()`, and so on) it needs to be the data model for the list, and can be used to broadcast model changes to multiple components.

If the array contains objects, the `List.labelField` or `List.labelFunction` properties are accessed to determine what parts of the item to display. The default value is `"label"`, so if a `label` field exists, it is chosen for display; if it doesn't exist, a comma-separated list of all fields is displayed.

Note: If the array contains strings at each index, and not objects, the list is not able to sort the items and maintain the selection state. Any sorting will cause the selection to be lost.

Any instance that implements the DataProvider API can be a data provider for a List component. This includes Flash Remoting recordsets, Firefly data sets, and so on.

Example

This example uses an array of strings to populate the list:

```
list.dataProvider = ["Ground Shipping","2nd Day Air","Next Day Air"];
```

This example creates a data provider array and assigns it to the `dataProvider` property, as in the following:

```
myDP = new Array();
list.dataProvider = myDP;

for (var i=0; i<accounts.length; i++) {
  // these changes to the data provider will be broadcast to the list
  myDP.addItem({label: accounts[i].name,
                data:  accounts[i].accountID});
}
```

List.getItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.getItemAt(*index*)

Parameters

index A number greater than or equal to 0, and less than `List.length`. It specifies the index of the item to retrieve.

Returns

The indexed item object; `undefined` if the index is out of range.

Description

Method; retrieves the item at the specified index.

Example

The following code displays the label of the item at index position 4:

```
trace(myList.getItemAt(4).label);
```

List.hPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.hPosition

Description

Property; scrolls the list horizontally to the number of pixels specified. You can't set `hPosition` unless the value of `hScrollPolicy` is `"on"` and the list has a `maxHPosition` that is greater than 0.

Example

The following example gets the horizontal scroll position of `myList`:

```
var scrollPos = myList.hPosition;
```

The following example sets the horizontal scroll position all the way to the left:

```
myList.hPosition = 0;
```

List.hScrollPolicy

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`listInstance``.hScrollPolicy`

Description

Property; a string that determines whether the horizontal scroll bar is displayed; the value can be `"on"` or `"off"`. The default value is `"off"`. The horizontal scroll bar does not measure text; you must set a maximum horizontal scroll position (see `List.maxHPosition`).

Note: `List.hScrollPolicy` does not support the value `"auto"`.

Example

The following code enables the list to scroll horizontally up to 200 pixels:

```
myList.hScrollPolicy = "on";
myList.Box.maxHPosition = 200;
```

See also

`List.hPosition`, `List.maxHPosition`

List.iconField

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`listInstance``.iconField`

Description

Property; specifies the name of a field to be used as an icon identifier. If the field has a value of `undefined`, the default icon specified by the `defaultIcon` style is used. If the `defaultIcon` style is `undefined`, no icon is used.

Example

The following example sets the `iconField` property to the `icon` property of each item:

```
list.iconField = "icon"
```

See also

`List.iconFunction`

List.iconFunction

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`listInstance``.iconFunction`

Description

Property; specifies a function that determines which icon each row will use to display its item. This function receives a parameter, *`item`*, which is the item being rendered, and must return a string representing the icon's symbol identifier.

Example

The following example adds icons that indicate whether a file is an image or a text document. If the `data.fileExtension` field contains a value of `"jpg"` or `"gif"`, the icon used will be `"pictureIcon"`, and so on.

```
list.iconFunction = function(item){
  var type = item.data.fileExtension;
  if (type=="jpg" || type=="gif") {
    return "pictureIcon";
  } else if (type=="doc" || type=="txt") {
    return "docIcon";
  }
}
```

List.itemRollOut

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(itemRollOut){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.itemRollOut = function(eventObject){
  // your code here
}
listInstance.addEventListener("itemRollOut", listenerObject)
```

Event object

In addition to the standard properties of the event object, the `itemRollOut` event has an `index` property, which specifies the number of the item that was rolled out.

Description

Event; broadcast to all registered listeners when the pointer rolls over and then off of list items.

The first usage example uses an `on()` handler and must be attached directly to a List instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the List instance `myList`, sends "_level0.myList" to the Output panel:

```
on(itemRollOut){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`listInstance`*) dispatches an event (in this case, `itemRollOut`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class (API)" on page 415.

Example

The following example sends a message to the Output panel that indicates which item index number has been rolled over:

```
form.itemRollOut = function (eventObj) {
  trace("Item #" + eventObj.index + " has been rolled out.");
}
myList.addEventListener("itemRollOut", form);
```

See also

List.itemRollOver

List.itemRollOver

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(itemRollOver){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.itemRollOver = function(eventObject){
  // your code here
}
listInstance.addEventListener("itemRollOver", listenerObject)
```

Event object

In addition to the standard properties of the event object, the `itemRollOver` event has an `index` property that specifies the number of the item that was rolled over.

Description

Event; broadcast to all registered listeners when the list items are rolled over.

The first usage example uses an `on()` handler and must be attached directly to a List instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the List instance `myList`, sends "_level0.myList" to the Output panel:

```
on(itemRollOver){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*listInstance*) dispatches an event (in this case, itemRollOver) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the EventDispatcher.addEventListener() method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class (API)" on page 415.

Example

The following example sends a message to the Output panel that indicates which item index number has been rolled over:

```
form.itemRollOver = function (eventObj) {
  trace("Item #" + eventObj.index + " has been rolled over.");
}
myList.addEventListener("itemRollOver", form);
```

See also

List.itemRollOut

List.labelField

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.labelField

Description

Property; specifies a field in each item to be used as display text. This property takes the value of the field and uses it as the label. The default value is "label".

Example

The following example sets the labelField property to be the "name" field of each item. "Nina" would display as the label for the item added in the second line of code:

```
list.labelField = "name";
list.addItem({name: "Nina", age: 25});
```

See also

List.labelFunction

List.labelFunction

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.labelFunction

Description

Property; specifies a function that determines which field (or field combination) of each item to display. This function receives one parameter, *item*, which is the item being rendered, and must return a string representing the text to display.

Example

The following example makes the label display some formatted details of the items:

```
list.labelFunction = function(item){
  return "The price of product " + item.productID + ", " + item.productName +
  " is $"
+ item.price;
}
```

See also

List.labelField

List.length

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.length

Description

Property (read-only); the number of items in the list.

Example

The following example places the value of `length` in a variable:

```
var len = myList.length;
```

List.maxHPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.maxHPosition

Description

Property; specifies the number of pixels the list can scroll when List.hScrollPolicy is set to "on". The list doesn't precisely measure the width of text that it contains. You must set maxHPosition to indicate the amount of scrolling that the list requires. The list does not scroll horizontally if this property is not set.

Example

The following example creates a list with 400 pixels of horizontal scrolling:

```
myList.hScrollPolicy = "on";
myList.maxHPosition = 400;
```

See also

List.hScrollPolicy

List.multipleSelection

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.multipleSelection

Description

Property; indicates whether multiple selections are allowed (true) or only single selections are allowed (false). The default value is false.

Example

The following example tests to determine whether multiple items can be selected:

```
if (myList.multipleSelection){
  // your code here
}
```

The following example allows the list to take multiple selections:

```
myList.multipleSelection = true;
```

List.removeAll()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.removeAll()
```

Parameters

None.

Returns

Nothing.

Description

Method; removes all items in the list.

Calling this method modifies the data provider of the List component. If the data provider is shared with other components, those components will update as well.

Example

The following code clears the list:

```
myList.removeAll();
```

List.removeItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.removeItemAt(index)
```

Parameters

index A string that indicates the label for the new item. The value must be greater than 0 and less than `List.length`.

Returns

An object; the removed item (`undefined` if no item exists).

Description

Method; removes the item at the specified index position. The list indices after the specified index collapse by one.

Calling this method modifies the data provider of the List component. If the data provider is shared with other components, those components will update as well.

Example

The following code removes the item at index position 3:

```
myList.removeItemAt(3);
```

List.replaceItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.replaceItemAt(index, label[, data])
listInstance.replaceItemAt(index, itemObject)
```

Parameters

index A number greater than 0 and less than `List.length` that indicates the position at which to insert the item (the index of the new item).

label A string that indicates the label for the new item.

data The data for the item. This parameter is optional and can be of any type.

itemObject An object to use as the item, usually containing `label` and `data` properties.

Returns

Nothing.

Description

Method; replaces the content of the item at the specified index.

Calling this method modifies the data provider of the List component. If the data provider is shared with other components, those components will update as well.

Example

The following example changes the fourth index position:

```
myList.replaceItemAt(3, "new label");
```

List.rowCount

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.rowCount

Description

Property; the number of rows that are at least partially visible in the list. This is useful if you've scaled a list by pixel and need to count its rows. Conversely, setting the number of rows guarantees that an exact number of rows will be displayed, without a partial row at the bottom.

The code `myList.rowCount = num` is equivalent to the code `myList.setSize(myList.width, h)` (where `h` is the height required to display `num` items).

The default value is based on the height of the list as set during authoring, or set by the `list.setSize()` method (see `UIObject.setSize()`).

Example

The following example discovers the number of visible items in a list:

```
var rowCount = myList.rowCount;
```

The following example makes the list display four items:

```
myList.rowCount = 4;
```

This example removes a partial row at the bottom of a list, if there is one:

```
myList.rowCount = myList.rowCount;
```

This example sets a list to the smallest number of rows it can fully display:

```
myList.rowCount = 1;
trace("myList has "+myList.rowCount+" rows");
```

List.rowHeight

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.rowHeight

Description

Property; the height, in pixels, of every row in the list. The font settings do not make the rows grow to fit, so setting the `rowHeight` property is the best way to make sure items are fully displayed. The default value is 20.

Example

The following example sets each row to 30 pixels:

```
myList.rowHeight = 30;
```

List.scroll

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(scroll){
  // your code here
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.scroll = function(eventObject){
  // your code here
}
listInstance.addEventListener("scroll", listenerObject)
```

Event object

Along with the standard event object properties, the `scroll` event has one additional property, `direction`. It is a string with two possible values, `"horizontal"` or `"vertical"`. For a ComboBox scroll event, the value is always `"vertical"`.

Description

Event; broadcast to all registered listeners when a list scrolls.

The first usage example uses an `on()` handler and must be attached directly to a List instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the List instance `myList`, sends "_level0.myList" to the Output panel:

```
on(scroll){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*listInstance*) dispatches an event (in this case, `scroll`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class (API)" on page 415.

Example

The following example sends the instance name of the component that generated the `change` event to the Output panel:

```
form.scroll = function(eventObj){
  trace("list scrolled");
}
myList.addEventListener("scroll", form);
```

List.selectable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.selectable

Description

Property; a Boolean value that indicates whether the list is selectable (`true`) or not (`false`). The default value is `true`.

List.selectedIndex

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.selectedIndex

Description

Property; the selected index of a single-selection list. The value is `undefined` if nothing is selected; the value is equal to the last item selected if there are multiple selections. If you assign a value to `selectedIndex`, any current selection is cleared and the indicated item is selected.

Using the `selectedIndex` property to change selection doesn't dispatch a `change` event. To dispatch the `change` event, use the following code:

```
myList.dispatchEvent({type:"change", target:myList});
```

Example

This example selects the item after the currently selected item. If nothing is selected, item 0 is selected.

```
var selIndex = myList.selectedIndex;
myList.selectedIndex = (selIndex==undefined ? 0 : selIndex+1);
```

See also

`List.selectedIndices`, `List.selectedItem`, `List.selectedItems`

List.selectedIndices

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`listInstance``.selectedIndices`

Description

Property; an array of indices of the selected items. Assigning this property replaces the current selection. Setting `selectedIndices` to a zero-length array (or `undefined`) clears the current selection. The value is `undefined` if nothing is selected.

The `selectedIndices` property reflects the order in which the items were selected. If you click the second item, then the third item, and then the first item, `selectedIndices` returns `[1,2,0]`.

Example

The following example retrieves the selected indices:

```
var selIndices = myList.selectedIndices;
```

The following example selects four items:

```
var myArray = new Array (1,4,5,7);
myList.selectedIndices = myArray;
```

See also

`List.selectedIndex`, `List.selectedItem`, `List.selectedItems`

List.selectedItem

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.selectedItem

Description

Property (read-only); an item object in a single-selection list. (In a multiple-selection list with multiple items selected, `selectedItem` returns the item that was most recently selected.) If there is no selection, the value is `undefined`.

Example

This example displays the selected label:

```
trace(myList.selectedItem.label);
```

See also

`List.selectedIndex`, `List.selectedIndices`, `List.selectedItems`

List.selectedItems

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.selectedItems

Description

Property (read-only); an array of the selected item objects. In a multiple-selection list, `selectedItems` lets you access the set of items selected as item objects.

Example

The following example retrieves an array of selected item objects:

```
var myObjArray = myList.selectedItems;
```

See also

`List.selectedIndex`, `List.selectedItem`, `List.selectedIndices`

List.setPropertiesAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.setPropertiesAt(index, styleObj)
```

Parameters

index A number greater than 0 or less than `List.length` indicating the index of the item to change.

styleObj An object that enumerates the properties and values to set.

Returns

Nothing.

Description

Method; applies the specified properties to the specified item. The supported properties are `icon` and `backgroundColor`.

Example

The following example changes the fourth item to black and gives it an icon:

```
myList.setPropertiesAt(3, {backgroundColor:0x000000, icon: "file"});
```

List.sortItems()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listInstance.sortItems(compareFunc)
```

Parameters

compareFunc A reference to a function. This function is used to compare two items to determine their sort order.

For more information, see `Array.sort()` in *Flash ActionScript Language Reference.*

Returns

The index at which the item was added.

Description

Method; sorts the items in the list by using the function specified in the *compareFunc* parameter.

Example

The following example sorts the items according to uppercase labels. Note that the `a` and `b` parameters that are passed to the function are items that have `label` and `data` properties.

```
myList.sortItems(upperCaseFunc);
function upperCaseFunc(a,b){
    return a.label.toUpperCase() > b.label.toUpperCase();
}
```

See also

`List.sortItemsBy()`

List.sortItemsBy()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

myList.sortItemsBy(*fieldName*, *optionsFlag*)

myList.sortItemsBy(*fieldName*, *order*)

Parameters

fieldName A string that specifies the name of the field to use for sorting. This value is usually `"label"` or `"data"`.

order A string that specifies whether to sort the items in ascending order (`"ASC"`) or descending order (`"DESC"`).

optionsFlag Lets you perform multiple sorts of different types on a single array without having to replicate the entire array or resort it repeatedly.

The following are possible values for *optionsFlag*:

- `Array.DESCENDING`, which sorts from highest to lowest.
- `Array.CASEINSENSITIVE`, which sorts without regard to case.
- `Array.NUMERIC`, which sorts numerically if the two elements being compared are numbers. If they aren't numbers, use a string comparison (which can be case-insensitive if that flag is specified).
- `Array.UNIQUESORT`, which returns an error code (0) instead of a sorted array if two objects in the array are identical or have identical sort fields.

- `Array.RETURNINDEXEDARRAY`, which returns an integer index array that is the result of the sort. For example, the following array would return the second line of code and the array would remain unchanged:

```
["a", "d", "c", "b"]
[0, 3, 2, 1]
```

You can combine these options into one value. For example, the following code combines options 3 and 1:

```
array.sort (Array.NUMERIC | Array.DESCENDING)
```

Returns

Nothing.

Description

Method; sorts the items in the list in the specified order, using the specified field name. If the *`fieldName`* items are a combination of text strings and integers, the integer items are listed first. The *`fieldName`* parameter is usually `"label"` or `"data"`, but you can specify any primitive data value.

This is the fastest way to sort data in a component. It also maintains the component's selection state. The `sortItemsBy()` method is fast because it doesn't run any ActionScript while sorting. The `sortItems()` method needs to run an ActionScript compare function, and is therefore slower.

Example

The following code sorts the items in the list `surnameMenu` in ascending order using the labels of the list items:

```
surnameMenu.sortItemsBy("label", "ASC");
```

See also

`List.sortItems()`

List.vPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`listInstance``.vPosition`

Description

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.

Example

The following example sets the position of the list to the first index item:

```
myList.vPosition = 0;
```

List.vScrollPolicy

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

listInstance.vScrollPolicy

Description

Property; a string that determines whether the list supports vertical scrolling. The value of this property can be `"on"`, `"off"` or `"auto"`. The value `"auto"` causes a scroll bar to appear when needed.

Example

The following example disables the scroll bar:

```
myList.vScrollPolicy = "off";
```

You can still create scrolling by using `List.vPosition`.

See also

`List.vPosition`

Loader component

The Loader component is a container that can display a SWF or JPEG file. You can scale the contents of the loader or resize the loader itself to accommodate the size of the contents. By default, the contents are scaled to fit the loader. You can also load content at runtime and monitor loading progress.

A Loader component can't receive focus. However, content loaded into the Loader component can accept focus and have its own focus interactions. For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each Loader instance reflects changes made to parameters in the Property inspector or Component inspector during authoring.

You can use the Accessibility panel to make Loader component content accessible to screen readers. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the Loader component

You can use a loader whenever you need to retrieve content from a remote location and pull it into a Flash application. For example, you could use a loader to add a company logo (JPEG file) to a form. You could also use a loader to leverage Flash work that has already been completed. For example, if you had already built a Flash application and wanted to expand it, you could use the loader to pull the old application into a new application, perhaps as a section of a tab interface. In another example, you could use the loader component in an application that displays photos. Use `Loader.load()`, `Loader.percentLoaded`, and `Loader.complete` to control the timing of the image loads and display progress bars to the user during loading.

If you load certain version 2 components into a SWF file or into the Loader component, the components may not work correctly. These components include the following: Alert, ComboBox, DateField, Menu, MenuBar, and Window.

Use the `_lockroot` property when calling `loadMovie()` or loading into the Loader component. If you're using the Loader component, add the following code:

```
myLoaderComponent.content._lockroot = true;
```

If you're using a movie clip with a call to `loadMovie()`, add the following code:

```
myMovieClip._lockroot = true;
```

If you don't set `_lockroot` to `true` in the loader movie clip, the loader only has access to its own library, but not the library in the loaded movie clip.

The `_lockroot` property is supported by Flash Player 7. For information about this property, see `MovieClip._lockroot` in *Flash ActionScript Language Reference*.

Loader parameters

You can set the following authoring parameters for each Loader component instance in the Property inspector or in the Component inspector:

autoload indicates whether the content should load automatically (`true`), or wait to load until the `Loader.load()` method is called (`false`). The default value is `true`.

contentPath an absolute or relative URL indicating the file to load into the loader. A relative path must be relative to the SWF file loading the content. The URL must be in the same subdomain as the URL where the Flash content currently resides. For use in Flash Player or in test-movie mode, all SWF files must be stored in the same folder, and the filenames cannot include folder or disk drive specifications. The default value is `undefined` until the load starts.

scaleContent indicates whether the content scales to fit the loader (`true`), or the loader scales to fit the content (`false`). The default value is `true`.

You can write ActionScript to set additional options for Loader instances using its methods, properties, and events. For more information, see "Loader class" on page 486.

Creating an application with the Loader component

The following procedure explains how to add a Loader component to an application while authoring. In this example, the loader loads a logo JPEG from an imaginary URL.

To create an application with the Loader component:

1. Drag a Loader component from the Components panel to the Stage.
2. Select the loader on the Stage and use the Free Transform tool to size it to the dimensions of the corporate logo.
3. In the Property inspector, enter the instance name **logo**.
4. Select the loader on the Stage and in the Component inspector, and enter **http://corp.com/websites/logo/corplogo.jpg** for the contentPath parameter.

Customizing the Loader component

You can transform a Loader component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`).

The sizing behavior of the Loader component is controlled by the `scaleContent` property. When `scaleContent` is `true`, the content is scaled to fit within the bounds of the loader (and is rescaled when `UIObject.setSize()` is called). When `scaleContent` is `false`, the size of the component is fixed to the size of the content and `UIObject.setSize()` has no effect.

Using styles with the Loader component

The Loader component uses the following styles.

Style	Theme	Description
`border styles`	Both	The Loader component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style is `"none"`.

Using skins with the Loader component

The Loader component uses an instance of RectBorder for its border (see "RectBorder class" on page 647).

Loader class

Inheritance MovieClip > UIObject class > UIComponent class > View > Loader

ActionScript Class Name mx.controls.Loader

The properties of the Loader class let you set content to load and monitor its loading progress at runtime.

Setting a property of the Loader class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.Loader.version);
```

Note: The code `trace(myLoaderInstance.version);` returns `undefined`.

Method summary for the Loader class

The following table lists the method of the Loader class.

Method	Description
`Loader.load()`	Loads the content specified by the `contentPath` property.

Methods inherited from the UIObject class

The following table lists the methods the Loader class inherits from the UIObject class. When calling these methods from the Loader object, use the form *`LoaderInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.

Method	Description
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Loader class inherits from the UIComponent class. When calling these methods from the Loader object, use the form *LoaderInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the Loader class

The following table lists properties of the Loader class.

Property	Description
Loader.autoLoad	A Boolean value that indicates whether the content loads automatically (true) or you must call Loader.load() (false).
Loader.bytesLoaded	A read-only property that indicates the number of bytes that have been loaded.
Loader.bytesTotal	A read-only property that indicates the total number of bytes in the content.
Loader.content	A reference to the content of the loader. This property is read-only.
Loader.contentPath	A string that indicates the URL of the content to be loaded.
Loader.percentLoaded	A number that indicates the percentage of loaded content. This property is read-only.
Loader.scaleContent	A Boolean value that indicates whether the content scales to fit the loader (true), or the loader scales to fit the content (false).

Properties inherited from the UIObject class

The following table lists the properties the Loader class inherits from the UIObject class. When accessing these properties from the Loader object, use the form *LoaderInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.

Property	Description
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Loader class inherits from the UIComponent class. When accessing these properties from the Loader object, use the form *LoaderInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the Loader class

The following table lists events of the Loader class.

Event	Description
Loader.complete	Triggered when the content finished loading.
Loader.progress	Triggered while content is loading.

Events inherited from the UIObject class

The following table lists the events the Loader class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.

Event	Description
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Loader class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Loader.autoLoad

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

loaderInstance.autoLoad

Description

Property; a Boolean value that indicates whether to automatically load the content (`true`), or wait until `Loader.load()` is called (`false`). The default value is `true`.

Example

The following code sets up the loader component to wait for a `Loader.load()` call:

```
loader.autoload = false;
```

Loader.bytesLoaded

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

loaderInstance.bytesLoaded

Description

Property (read-only); the number of bytes of content that have been loaded. The default value is 0 until content begins loading.

Example

The following code creates a progress bar and a Loader component. It then creates a listener object with a `progress` event handler that shows the progress of the load. The listener is registered with the `loader` instance.

```
createClassObject(mx.controls.ProgressBar, "pBar", 0);
createClassObject(mx.controls.Loader, "loader", 1);
loadListener = new Object();
loadListener.progress = function(eventObj){
  // eventObj.target is the component that generated the progress event,
  // that is, the loader
  pBar.setProgress(loader.bytesLoaded, loader.bytesTotal); // show progress
}
loader.addEventListener("progress", loadListener);
loader.content = "logo.swf";
```

When you create an instance with `createClassObject()`, you have to position it on the Stage with `move()` and `setSize()`. See `UIObject.move()` and `UIObject.setSize()`.

See also

`Loader.bytesTotal`, `UIObject.createClassObject()`

Loader.bytesTotal

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`loaderInstance``.bytesTotal`

Description

Property (read-only); the size of the content, in bytes. The default value is 0 until content begins loading.

Example

The following code creates a progress bar and a Loader component. It then creates a load listener object with a `progress` event handler that shows the progress of the load. The listener is registered with the loader instance, as follows:

```
createClassObject(mx.controls.ProgressBar, "pBar", 0);
createClassObject(mx.controls.Loader, "loader", 1);
loadListener = new Object();
loadListener.progress = function(eventObj){
```

```
  // eventObj.target is the component that generated the progress event,
  // that is, the loader
  pBar.setProgress(loader.bytesLoaded, loader.bytesTotal); // show progress
}
loader.addEventListener("progress", loadListener);
loader.content = "logo.swf";
```

See also

Loader.bytesLoaded

Loader.complete

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(complete){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.complete = function(eventObject){
  ...
}
loaderInstance.addEventListener("complete", listenerObject)
```

Description

Event; broadcast to all registered listeners when the content has finished loading.

The first usage example uses an `on()` handler and must be attached directly to a Loader instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the Loader instance `myLoaderComponent`, sends "_level0.myLoaderComponent" to the Output panel:

```
on(complete){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*loaderInstance*) dispatches an event (in this case, `complete`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example creates a Loader component and then defines a listener object with a `complete` event handler that sets the loader's `visible` property to `true`:

```
createClassObject(mx.controls.Loader, "loader", 0);
loadListener = new Object();
loadListener.complete = function(eventObj){
  loader.visible = true;
}
loader.addEventListener("complete", loadListener);
loader.contentPath = "logo.swf";
```

Loader.content

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

loaderInstance.content

Description

Property (read-only); a reference to a movie clip instance that contains the contents of the loaded file. The value is `undefined` until the load begins.

See also

`Loader.contentPath`

Loader.contentPath

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

loaderInstance.contentPath

Description

Property; a string that indicates an absolute or relative URL of the file to load into the loader. A relative path must be relative to the SWF file that loads the content. The URL must be in the same subdomain as the loading SWF file.

If you are using Flash Player or test-movie mode in Flash, all SWF files must be stored in the same folder, and the filenames cannot include folder or disk drive information.

Example

The following example tells the loader instance to display the contents of the logo.swf file:

```
loader.contentPath = "logo.swf";
```

Loader.load()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

loaderInstance.load([*path*])

Parameters

path An optional parameter that specifies the value for the contentPath property before the load begins. If a value is not specified, the current value of contentPath is used as is.

Returns

Nothing.

Description

Method; tells the loader to begin loading its content.

Example

The following code creates a Loader instance and sets the `autoload` property to `false` so that the loader must wait for a call to `load()` to begin loading content. Next, the `contentPath` property is set, which indicates where to load content from. Then other tasks can be performed before the content is loaded with `loader.load()`.

```
createClassObject(mx.controls.Loader, "loader", 0);
loader.autoLoad = false;
loader.contentPath = "logo.swf";

// Perform other tasks here and *then* start loading the file.

loader.load();
```

Loader.percentLoaded

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`loaderInstance``.percentLoaded`

Description

Property (read-only); a number indicating what percent of the content has loaded. Typically, this property is used to present the progress to the user in an easily readable form. Use the following code to round the figure to the nearest integer:

```
Math.round(bytesLoaded/bytesTotal*100))
```

Example

The following example creates a Loader instance and then creates a listener object with a `progress` handler that traces the percent loaded and sends it to the Output panel:

```
createClassObject(Loader, "loader", 0);
loadListener = new Object();
loadListener.progress = function(eventObj){
  // eventObj.target is the component that generated the progress event,
  // that is, the loader
  trace("logo.swf is " + loader.percentLoaded + "% loaded."); // track loading
   progress
}
loader.addEventListener("complete", loadListener);
loader.content = "logo.swf";
```

Loader.progress

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(progress){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.progress = function(eventObject){
  ...
}
loaderInstance.addEventListener("progress", listenerObject)
```

Description

Event; broadcast to all registered listeners while content is loading. This event occurs when the load is triggered by the `autoload` parameter or by a call to `Loader.load()`. The progress event is not always broadcast, and the `complete` event may be broadcast without any `progress` events being dispatched. This can happen if the loaded content is a local file.

The first usage example uses an `on()` handler and must be attached directly to a Loader instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the Loader instance `myLoaderComponent`, sends "_level0.myLoaderComponent" to the Output panel:

```
on(progress){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`loaderInstance`*) dispatches an event (in this case, `progress`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following code creates a Loader instance and then creates a listener object with an event handler for the `progress` event that sends a message to the Output panel telling what percent of the content has loaded:

```
createClassObject(mx.controls.Loader, "loader", 0);
loadListener = new Object();
loadListener.progress = function(eventObj){
  // eventObj.target is the component that generated the progress event,
  // that is, the loader
  trace("logo.swf is " + loader.percentLoaded + "% loaded."); // track loading
   progress
}
loader.addEventListener("progress", loadListener);
loader.contentPath = "logo.swf";
```

Loader.scaleContent

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

loaderInstance.scaleContent

Description

Property; indicates whether the content scales to fit the loader (true), or the loader scales to fit the content (`false`). The default value is `true`.

Example

The following code tells the loader to resize itself to match the size of its content:

```
loader.scaleContent = false;
```

Media components (Flash Professional only)

The streaming media components make it easy to incorporate streaming media into Flash presentations. These components let you present your media in a variety of ways.

You can use the following three media components:

- The MediaDisplay component lets media stream into your Flash content without a supporting user interface. You can use this component with video and audio data. When you use this component by itself, the user has no control over the media.
- The MediaController component provides standard user interface controls (play, pause, and so on) for media playback. Media is never loaded into or played by this component; it is used only for controlling playback in a MediaPlayback or MediaDisplay instance. The MediaController component features a "drawer," which displays the contents of the playback controls when the mouse is positioned over the component.
- The MediaPlayback component is a combination of the MediaDisplay and MediaController components; it provides methods to stream your media content.

Bear in mind these points about media components:

- The media components require Flash Player 6 or later. In Flash Player 6, media components support FLV files only through Flash Remoting, not through HTTP.
- The media components do not support scan forward and scan backward functionality. However, you can effect this functionality by moving the playback slider.
- Only component size and controller policy are reflected in the live preview.
- The media components do not support accessibility.

Interacting with media components (Flash Professional only)

The streaming MediaPlayback and MediaController components respond to mouse and keyboard activity; the MediaDisplay component does not. The following table summarizes the actions for the MediaPlayback and MediaController components upon receiving focus:

Target	Navigation	Description
Playback controls of a given controller	Mouse rollover	The button is highlighted.
Playback controls of a given controller	Single click of left mouse button	Users can click the playback controls to manipulate the playback of audio and video media. The Pause/Play and Go to Beginning/Go to End buttons behave as standard buttons. When the mouse button is pressed, the onscreen button highlights in its pressed state, and when the mouse button is released, the onscreen button reverts to its unselected appearance. The Go to End button is disabled when FLV media files are playing.

Target	Navigation	Description
Slider controls of a given controller	Move slider back and forth	The playbar indicates the user's position within the media; the playback slider moves horizontally (by default) to indicate the playback from beginning (left) to end (right). The slider moves from bottom to top when the controls are oriented vertically. As the slider moves from left to right, it highlights the previous display space to indicate that this content has been played back or selected. Display space ahead of the slider remains unhighlighted until the slider passes. Users can drag the slider to affect the media's playback position. If media is playing, automatic playback begins from the point at which the mouse is released. If the media is paused, the user can move and release the slider and the media remains paused. There is also a volume slider, which moves from left (muted) to right (maximum volume) in both the horizontal and vertical layouts.
Playback controller navigation	Tab, Shift+Tab	Moves the focus from button to button within the controller component, where the focused element becomes highlighted. This navigation works with the Pause/Play, Go to Beginning, Go to End, Volume Mute, and Volume Max controls. The focus moves from left to right and top to bottom as users tab through the elements. Shift+Tab moves focus from right to left and bottom to top. Upon receiving focus through the Tab key, the control immediately passes focus to the Play/Pause button. When focus is on the Volume Max button, and then Tab is pressed, the focus moves to the next control in the tab index on the Stage.
A given control button	Spacebar or Enter/Return	Selects the element in focus. On press, the button appears in its pressed state. On release, the button reverts back to its focused, mouse-over state.

Understanding media components (Flash Professional only)

This section provides an overview of how the media components work. Most of the properties listed in this section can be set with the Component inspector. (See "Using the Component inspector with media components" on page 503.)

Apart from the layout properties discussed later in this section, the following properties can be set for the MediaDisplay and MediaPlayback components:

- The media type, which can be set to MP3 or FLV (see `Media.mediaType` and `Media.setMedia()`).
- The relative or absolute content path, which holds the media file to be streamed (see `Media.contentPath`).

- Cue point objects, along with their name, time, and player properties (see `Media.addCuePoint()` and `Media.cuePoints`). The name of the cue point is arbitrary; use a name that will have meaning when using listener and trace events. A cue point broadcasts a `cuePoint` event when the value of its time property is equal to that of the playhead location of the MediaPlayback or MediaDisplay component with which it is associated. The player property is a reference to the MediaPlayback instance with which it is associated. You can remove cue points by using `Media.removeCuePoint()` and `Media.removeAllCuePoints()`.

The streaming media components broadcast several related events. The following broadcast events can be used to set other items in motion:

- A `change` event is broadcast continuously by the MediaDisplay and MediaPlayback components while media is playing. (See `Media.change`.)
- A `progress` event is continuously broadcast by the MediaDisplay and MediaPlayback components while media is loading. (See `Media.progress`.)
- A `click` event is broadcast by the MediaController and MediaPlayback components whenever the user clicks the Pause/Play button. In this case, the `detail` property of the event object provides information on which button was clicked. (See `Media.click`.)
- A `volume` event is broadcast by the MediaController and MediaPlayback components when the user adjusts the volume controls. (See `Media.volume`.)
- A `playheadChange` event is broadcast by the MediaController and MediaPlayback components when the user moves the playback slider or when the Go to Beginning or Go to End buttons are clicked. (See `Media.playheadChange`.)

The MediaDisplay component works with the MediaController component. Combined, the components behave in a manner similar to the MediaPlayback component, but they give you more flexibility in the look and feel of your media presentation.

Understanding the MediaDisplay component

When you place a MediaDisplay component on the Stage, it has no user interface. It is simply a container to hold and play media. The following properties affect the appearance of video media playing in a MediaDisplay component:

- `Media.aspectRatio`
- `Media.autoSize`
- Height (in the Property inspector)
- Width (in the Property inspector)

Note: The user does not see anything unless some media is playing.

The `Media.aspectRatio` property takes precedence over the other properties. When `Media.aspectRatio` is set to `true` (the default), the component always readjusts the size of the playing media to maintain the media's aspect ratio.

For FLV files, when `Media.autoSize` is set to `true`, the media is displayed at its preferred size, regardless of the size of the component. This means that if the size of the MediaDisplay instance size is different from the size of the media, the media will either spill out of the instance boundaries or not fill the instance size. When `Media.autoSize` is set to `false`, Flash uses the instance size as much as possible, while honoring the aspect ratio. If both `Media.autoSize` and `Media.aspectRatio` are set to `false`, the exact size of the component is used.

Note: Since there is no image to show with MP3 files, setting `Media.autoSize` would have no effect. For MP3 files, the minimum usable size is 60 pixels high by 256 pixels wide in the default orientation.

The MediaDisplay component also supports the `Media.volume` property. This property takes on integer values from 0 (mute) to 100 (maximum volume). The default setting is 75.

Understanding the MediaController component

The interface for the MediaController component depends on its `Media.controllerPolicy` and `Media.backgroundStyle` properties. The `Media.controllerPolicy` property determines if the media control set is always visible, collapsed, or only visible when the mouse hovers over the control portion of the component. When collapsed, the controller draws a modified progress bar, which is a combination of the loadbar and the playbar. It shows the progress of the bytes being loaded at the bottom of the bar, and the progress of the playhead just above it. When expanded, the controller draws an enhanced version of the playbar/loadbar, which contains the following items:

- Text labels on the left that indicate the playback state (streaming or paused), and on the right that indicate playhead location, in seconds
- A playhead location indicator
- A slider, which users can drag to navigate through the media

The MediaController component also provides the following items:

- A Play/Pause button
- Go to Beginning and Go to End buttons, which navigate to the beginning and end of the media, respectively
- A volume control that consists of a slider, a mute button, and a maximum volume button

Both the collapsed and expanded states of the MediaController component use the `Media.backgroundStyle` property. This property determines whether the controller draws a chrome background (the default) or allows the media background to display from behind the controls.

The MediaController component has an orientation setting, `Media.horizontal`, which you can use to draw the component with a horizontal orientation (the default) or a vertical one. With a horizontal orientation, the playbar tracks playing media from left to right. With a vertical orientation, the playbar tracks media from bottom to top.

You can associate the MediaDisplay and MediaController components with each other by using the `Media.associateDisplay()` and `Media.associateController()` methods. These methods allow the MediaController instance to update its controls based on events broadcast from the MediaDisplay instance, and allow the MediaDisplay component to react to user settings in the MediaController.

Understanding the MediaPlayback component

The MediaPlayback contains the MediaController and MediaDisplay subcomponents. The MediaController and MediaDisplay portions always scale to fit the size of the overall MediaPlayback instance.

The MediaPlayback component uses `Media.controlPlacement` to determine the layout of the controls. By setting this property to `top`, `bottom`, `left`, or `right`, you can indicate 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.

Using media components (Flash Professional only)

With the sharp increase in the use of media to provide information to web users, many developers want their users to be able to stream media and then control it. You might use media components in the following kinds of situations:

- Showing media that introduces a company
- Streaming movies or movie previews
- Streaming songs or song snippets
- Providing learning material in the form of media

Using the MediaPlayback component

Suppose you must develop a website that allows users to preview DVDs and CDs that you sell in a rich media environment. The following example shows the steps involves. (It assumes your website is ready for inserting streaming components.)

To create a Flash document that displays a CD or DVD preview:

1. Select File > New; then select Flash Document.
2. Open the Components panel and double-click the MediaPlayback component to place an instance of it on the Stage.
3. Select the MediaPlayback component instance and enter the instance name **myMedia** in the Property inspector.
4. In the Component inspector, set your media type according to the type of media that will be streaming (MP3 or FLV).
5. If you selected FLV, enter the duration of the video in the Video Length text boxes; use the format *HH:MM:SS*.
6. Enter the location of your preview video in the URL text box. For example, you might enter **http://my.web.com/videopreviews/AMovieName.flv**.

7. Set the desired options for the Automatically Play, Use Preferred Media Size, and Respect Aspect Ratio check boxes.
8. Set the control placement to the desired side of the MediaPlayback component.
9. Add a cue point toward the end of the media; this cue point will be used with a listener to open a pop-up window that announces that the movie is on sale. Give the cue point the name **cuePointName**.

   Next, you'll set the cue point time such that it is within a few seconds of the end of the clip.
10. Double-click a Window component to place on the Stage; then delete it.

    This places a symbol called Window in your library.
11. Create a text box and write some text informing the user that the movie is on sale.
12. Select Modify > Convert to Symbol to convert the text box to a movie clip, and give it the name **mySale_mc**.
13. Right-click (Windows) or Control-click (Macintosh) the `mySale_mc` movie clip in the library, select Linkage, and select Export for ActionScript.

    This places the movie clip in your runtime library.
14. Add the following ActionScript to Frame 1. This code creates a listener to open a pop-up window informing the user that the movie is on sale.

```
// Import the classes necessary to create the pop-up window dynamically

import mx.containers.Window;
import mx.managers.PopUpManager;

// Create a listener object to open sale pop-up
var saleListener = new Object();

saleListener.cuePoint = function(evt){

var saleWin = PopUpManager.createPopUp(_root, Window, false, {closeButton:
  true, title: "Movie Sale ", contentPath: "mySale_mc"});

// Enlarge the window so that the content fits

saleWin.setSize(80, 80);
var delSaleWin = new Object();
delSaleWin.click = function(evt){
saleWin.deletePopUp();
}
saleWin.addEventListener("click", delSaleWin);

}

myMedia.addEventListener("cuePoint", saleListener);
```

Using the MediaDisplay and MediaController components

If you want a lot of control over the look and feel of your media display, you may want to use the MediaDisplay and MediaController components together. The following example creates a Flash application that displays your CD and DVD preview media.

To create a Flash document that displays a CD or DVD preview:

1. In Flash, select File > New; then select Flash Document.
2. In the Components panel, double-click the MediaController and MediaDisplay components to place instances on the Stage.
3. Select the MediaDisplay instance and enter the instance name **myDisplay** in the Property inspector.
4. Select the MediaController instance and enter the instance name **myController** in the Property inspector.
5. Open the Component inspector from the Property inspector and set your media type according to the type of media that will be streaming (MP3 or FLV).
6. If you selected FLV, enter the duration of the video in the Video Length text boxes using the format *HH:MM:SS*.
7. Enter the location of your preview video in the URL text box. For example, you might enter **http://my.web.com/videopreviews/AMovieName.flv**.
8. Set the desired options for the Automatically Play, Use Preferred Media Size, and Respect Aspect Ratio check boxes.
9. Select the MediaController instance and, in the Component inspector, set your orientation to vertical by setting the `horizontal` property to `false`.
10. In the Component inspector, set `backgroundStyle` to None.

    This specifies that the MediaController instance should not draw a background but should instead fill the media between the controls.

    Next, you'll use a behavior to associate the MediaController and MediaDisplay instances so that the MediaController instance accurately reflects the playhead movement and other settings in the MediaDisplay instance, and so that the MediaDisplay instance responds to user clicks.
11. Select the MediaDisplay instance and, in the Property inspector, enter the instance name **myMediaDisplay**.
12. Select the MediaController instance that will trigger the behavior. In the Behaviors panel, click the Add (+) button and select Media > Associate Display.
13. In the Associate Display window, select `myMediaDisplay` under `_root` and click OK.

For more information on using behaviors with media components, see "Controlling media components by using behaviors" on page 504.

Using the Component inspector with media components

The Component inspector makes it easy to set media component parameters, properties, and so on. To use this panel, click the desired component on the Stage and, with the Property inspector open, click Launch Component Inspector. The Component inspector can be used for the following purposes:

- To automatically play the media (see `Media.activePlayControl` and `Media.autoPlay`)
- To keep or ignore the media's aspect ratio (see `Media.aspectRatio`)
- To determine if the media will be automatically sized to fit the component instance (see `Media.autoSize`)
- To enable or disable the chrome background (see `Media.backgroundStyle`)
- To specify the path to your media in the form of a URL (see `Media.contentPath`)
- To specify the visibility of the playback controls (see `Media.controllerPolicy`)
- To add cue point objects (see `Media.addCuePoint()`)
- To delete cue point objects (see `Media.removeCuePoint()`)
- To set the orientation of MediaController instances (see `Media.horizontal`)
- To set the type of media being played (see `Media.setMedia()`)
- To set the play time of the FLV media (see `Media.totalTime`)
- To set the last few digits of the time display to indicate milliseconds or frames per second (fps)

It is important to understand a few concepts when working with the Component inspector:

- The video time control is not available when you select an MP3 video type, because this information is automatically read in when MP3 files are used. For FLV files created with Flash Video Exporter 1.0, you must enter the total time of the media (`Media.totalTime`) in order for the playbar of the MediaPlayback component (or any listening MediaController component) to accurately reflect play progress. FLV files created with Flash Video Exporter 1.1 or later set the duration automatically.
- With the file type set to FLV, you'll notice a Milliseconds option and (if Milliseconds is unselected) a Frames Per Second (FPS) pop-up menu. When Milliseconds is selected, the FPS control is not visible. In this mode, the time displayed in the playbar at runtime is formatted as *HH:MM:SS.mmm* (*H* = hours, *M* = minutes, *S* = seconds, *m* = milliseconds), and cue points are set in seconds. When Milliseconds is unselected, the FPS control is enabled and the playbar time is formatted as *HH:MM:SS.FF* (*F* = frames per second), while cue points are set in frames.

  Note: You can set the FPS value only by using the Component inspector. Setting an fps value by using ActionScript has no effect and is ignored.

Controlling media components by using behaviors

Behaviors are prewritten ActionScript scripts that you add to an instance, such as a MediaDisplay component, to control that object. Behaviors let you add the power, control, and flexibility of ActionScript coding to your document without having to create the ActionScript code yourself.

To control a media component with a behavior, you use the Behaviors panel to apply the behavior to a given media component instance. You specify the event that will trigger the behavior (such as reaching a specified cue point), select a target object (the media components that will be affected by the behavior), and, if necessary, select settings for the behavior (such as the movie clip within the media to navigate to).

The following behaviors are packaged with Flash MX Professional 2004 and are used to control embedded media components.

Behavior	Purpose	Parameters
Associate Controller	Associates a MediaController component with a MediaDisplay component	Instance name of target MediaController components
Associate Display	Associates a MediaDisplay component with a MediaController component	Instance name of target MediaController components
Labeled Frame CuePoint Navigation	Places an action on a MediaDisplay or MediaPlayback instance that tells an indicated movie clip to navigate to a frame with the same name as a given cue point	Name of frame and name of cue point (the names should be equal)
Slide CuePoint Navigation	Makes a slide-based Flash document navigate to a slide with the same name as a given cue point	Name of slide and name of cue point (the names should be equal)

To associate a MediaDisplay component with a MediaController component:

1. Place a MediaDisplay instance and a MediaController instance on the Stage.
2. Select the MediaDisplay instance and, using the Property inspector, enter the instance name **myMediaDisplay**.
3. Select the MediaController instance that will trigger the behavior.
4. In the Behaviors panel, click the Add (+) button and select Media > Associate Display.
5. In the Associate Display window, select `myMediaDisplay` under `_root` and click OK.

Note: If you have associated the MediaDisplay component with the MediaController component, you do not need to associate the MediaController component with the MediaDisplay component.

To associate a MediaController component with a MediaDisplay component:

1. Place a MediaDisplay instance and a MediaController instance on the Stage.
2. Select the MediaController instance and, using the Property inspector, enter the instance name **myMediaController**.
3. Select the MediaDisplay instance that will trigger the behavior.
4. In the Behaviors panel, click the Add (+) button and select Media > Associate Controller.
5. In the Associate Controller window, select `myMediaController` under `_root` and click OK.

To use a Labeled Frame CuePoint Navigation behavior:

1. Place a MediaDisplay or MediaPlayback component instance on the Stage.
2. Select the desired frame that you want the media to navigate to and, using the Property inspector, enter the frame name **myLabeledFrame**.
3. Select your MediaDisplay or MediaPlayback instance.
4. In the Component inspector, click the Add (+) button and enter the cue point time in the format *HH:MM:SS:mmm* or *HH:MM:SS:FF*, and give the cue point the name **myLabeledFrame**.

   The cue point indicates the amount of time that should elapse before you navigate to the selected frame. For example, if you want to jump to `myLabeledFrame` 5 seconds into the media, enter **5** in the SS text box and enter **myLabeledFrame** in the Name text box.
5. In the Behaviors panel, click the Add (+) button and select Media > Labeled Frame CuePoint Navigation.
6. In the Labeled Frame CuePoint Navigation window, select the `_root` clip and click OK.

To use a Slide CuePoint Navigation behavior:

1. Open your new document as a Flash slide presentation.
2. Place a MediaDisplay or MediaPlayback component instance on the Stage.
3. In the Screen Outline pane to the left of the Stage, click the Insert Screen (+) button to add a second slide; then select the second slide and rename it **mySlide**.
4. Select your MediaDisplay or MediaController instance.
5. In the Component inspector, click the Add (+) button and enter the cue point time in the format *HH:MM:SS:mmm* or *HH:MM:SS:FF*, and give the cue point the name **MySlide**.

   The cue point indicates the amount of time that should elapse before you navigate to the selected slide. For example, if you want to jump to `mySlide` 5 seconds into the media, enter **5** in the SS text box and enter **mySlide** in the Name text box.
6. In the Behaviors panel, click the Add (+) button and select Media > Slide CuePoint Navigation.
7. In the Slide CuePoint Navigation window, select `Presentation` under the `_root` clip and click OK.

Media component parameters (Flash Professional only)

The following tables list authoring parameters that you can set for a given media component instance in the Property inspector.

MediaDisplay parameters

Name	Type	Default value	Description
Automatically Play (`Media.autoPlay`)	Boolean	Selected	Determines if the media plays as soon as it has loaded.
Use Preferred Media Size (`Media.autoSize`)	Boolean	Selected	Determines whether the media associated with the MediaDisplay instance conforms to the component size or simply uses its default size.
FPS	Integer	30	Indicates the number of frames per second. When the Milliseconds option is selected, this control is disabled.
Cue Points (`Media.cuePoints`)	Array	Undefined	An array of cue point objects, each with a name and position in time in a valid *HH:MM:SS:FF* (Milliseconds option selected) or *HH:MM:SS:mmm* format.
FLV or MP3 (`Media.mediaType`)	FLV or MP3	`FLV`	Designates the type of media to be played.
Milliseconds	Boolean	Unselected	Determines whether the playbar uses frames or milliseconds, and whether the cue points use seconds or frames. When this option is selected, the FPS control is not visible.
URL (`Media.contentPath`)	String	Undefined	A string that holds the path and filename of the media to be played.
Video Length (`Media.totalTime`)	Integer	Undefined	The total time needed to play the FLV media. This setting is required in order for the playbar to work correctly. This control is only visible when the media type is set to FLV.

MediaController parameters

Name	Type	Default value	Description
activePlayControl (`Media.activePlayControl`)	String: `pause` or `play`	`pause`	Determines whether the playbar is in play or pause mode upon instantiation. This mode determines the image displayed on the Play/Pause button, which is the opposite of the playing/paused state that the controller is actually in.
backgroundStyle (`Media.backgroundStyle`)	string: `default` or `none`	`default`	Determines whether the chrome background will be drawn for the MediaController instance.
controllerPolicy (`Media.controllerPolicy`)	`auto`, `on`, or `off`	`auto`	Determines whether the controller opens or closes according to mouse position, or is locked in the open or closed state.

Name	Type	Default value	Description
horizontal (`Media.horizontal`)	Boolean	`true`	Determines whether the controller portion of the instance is vertically or horizontally oriented. A `true` value indicates that the component will have a horizontal orientation.
enabled	Boolean	`true`	Determines whether this control can be modified by the user. A `true` value indicates that the control can be modified.
visible	Boolean	`true`	Determines whether this control is viewable by the user. A `true` value indicates that the control is viewable.
minHeight	Integer	0	Minimum height allowable for this instance, in pixels.
minWidth	Integer	0	Minimum width allowable for this instance, in pixels.

MediaPlayback parameters

Name	Type	Default value	Description
Control Placement (`Media.controlPlacement`)	`top`, `bottom`, `left`, `right`	`bottom`	Position of the controller. The value is related to orientation.
Control Visibility (`Media.controllerPolicy`)	Boolean	`true`	Determines whether the controller opens or closes according to mouse position.
Automatically Play (`Media.autoPlay`)	Boolean	Selected	Determines if the media plays as soon as it has loaded.
Use Preferred Media Size (`Media.autoSize`)	Boolean	Selected	Determines whether the MediaController instance sizes to fit the media or uses other settings.
FPS	Integer	30	Number of frames per second. When the Milliseconds option is selected, this control is disabled.
Cue Points (`Media.cuePoints`)	Array	Undefined	An array of cue point objects, each with a name and position in time in a valid *HH:MM:SS:mmm* (Milliseconds option selected) or *HH:MM:SS:FF* format.
FLV or MP3 (`Media.mediaType`)	FLV or MP3	`FLV`	Designates the type of media to be played.
Milliseconds	Boolean	Unselected	Determines whether the playbar uses frames or milliseconds, and whether the cue points use seconds or frames. When this option is selected, the FPS control is disabled.

Name	Type	Default value	Description
URL (`Media.contentPath`)	String	Undefined	A string that holds the path and filename of the media to be played.
Video Length (`Media.totalTime`)	Integer	Undefined	The total time needed to play the FLV media. This setting is required in order for the playbar to work correctly.

Creating applications with media components (Flash Professional only)

Creating Flash content by using media components is quite simple and often requires only a few steps. This example shows how to create an application to play a small, publicly available media file.

To add a media component to an application:

1. In Flash, select File > New; then select Flash Document.
2. In the Components panel, double-click the MediaPlayback component to add it to the Stage.
3. In the Property inspector, do the following:
   - Enter the instance name **myMedia**.
   - Click Launch Component Inspector.
4. In the Component inspector, enter **http://www.cathphoto.com/c.flv** in the URL text box.
5. Select Control > Test Movie to see the media play.

Customizing media components (Flash Professional only)

If you want to change the appearance of your media components, you can use skinning. For a complete guide to component customization, see Chapter 5, "Customizing Components," on page 67.

Using styles with media components

The media components do not use styles.

Using skins with media components

The media components do not support dynamic skinning, although you can open the media component source document and change their assets to achieve the desired look. It is best to make a copy of this file and work from the copy, so that you always have the installed source to go back to. You can find the media component source document at the following locations:

- Windows: C:\Documents and Settings*user*\Local Settings\Application Data\Macromedia\Flash MX 2004*language*\Configuration\ComponentFLA\MediaComponents.fla
- Macintosh: HD Drive/Users/*username*/Library/Application Support/Macromedia/Flash MX 2004/*language*/Configuration/ComponentFLA/MediaComponents.fla

Media class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > Media

ActionScript Class Names mx.controls.MediaController, mx.controls.MediaDisplay, mx.controls.MediaPlayback

Each component class has a `version` property, which is a class property. Class properties are available only for the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.MediaPlayback.version);
```

Note: The code `trace(myMediaInstance.version);` returns `undefined`.

Method summary for the Media class

The following table lists methods of the Media class.

Method	Components	Description
`Media.addCuePoint()`	MediaDisplay, MediaPlayback	Adds a cue point object to the component instance.
`Media.associateController()`	MediaDisplay	Associates a MediaDisplay instance with a MediaController instance.
`Media.associateDisplay()`	MediaController	Associates a MediaController instance with a MediaDisplay instance.
`Media.displayFull()`	MediaPlayback	Converts the component instance to full-screen playback mode.
`Media.displayNormal()`	MediaPlayback	Converts the component instance back to its original screen size.
`Media.getCuePoint()`	MediaDisplay, MediaPlayback	Returns a cue point object.
`Media.pause()`	MediaDisplay, MediaPlayback	Pauses the playhead at its current location in the media Timeline.
`Media.play()`	MediaDisplay, MediaPlayback	Plays the media associated with the component instance at a given starting point.
`Media.removeAllCuePoints()`	MediaDisplay, MediaPlayback	Deletes all cue point objects associated with a given component instance.
`Media.removeCuePoint()`	MediaDisplay, MediaPlayback	Deletes a specified cue point associated with a given component instance.
`Media.setMedia()`	MediaDisplay, MediaPlayback	Sets the media type and path to the specified media type.
`Media.stop()`	MediaDisplay, MediaPlayback	Stops the playhead and moves it to position 0, which is the beginning of the media.

Property summary for the Media class

The following table lists properties of the Media class.

Property	Components	Description
Media.activePlayControl	MediaController	Determines the component state when loaded at runtime.
Media.aspectRatio	MediaDisplay, MediaPlayback	Determines if the component instance maintains its video aspect ratio.
Media.autoPlay	MediaDisplay, MediaPlayback	Determines if the component instance immediately starts to buffer and play.
Media.autoSize	MediaDisplay, MediaPlayback	Determines the size of the media-viewing portion of the MediaDisplay or MediaPlayback component.
Media.backgroundStyle	MediaController	Determines if the component instance draws its chrome background.
Media.bytesLoaded	MediaDisplay, MediaPlayback	Read-only; the number of bytes loaded that are available for playing.
Media.bytesTotal	MediaDisplay, MediaPlayback	The number of bytes to be loaded into the component instance.
Media.contentPath	MediaDisplay, MediaPlayback	A string that holds the relative path and filename of the media to be streamed and played.
Media.controllerPolicy	MediaController, MediaPlayback	Determines whether the controller is hidden when instantiated and only appears when the user moves the mouse over the controller's collapsed state.
Media.controlPlacement	MediaPlayback	Determines where the component's controls are positioned.
Media.cuePoints	MediaDisplay, MediaPlayback	An array of cue point objects that have been assigned to a given component instance.
Media.horizontal	MediaController	Determines the orientation of the component instance.
Media.mediaType	MediaDisplay, MediaPlayback	Determines the type of media to be played.
Media.playheadTime	MediaDisplay, MediaPlayback	Holds the current position of the playhead (in seconds) for the media Timeline that is playing.
Media.playing	MediaDisplay, MediaPlayback, MediaController	For MediaDisplay and MediaPlayback, this property is read-only and returns a Boolean value to indicate whether a given component instance is playing media. For MediaController, this property is read/write and controls the image (playing or paused) displayed on the Play/Pause button of the controller.
Media.preferredHeight	MediaDisplay, MediaPlayback	The default value of the height of a FLV file.

Property	Components	Description
Media.preferredWidth	MediaDisplay, MediaPlayback	The default value of the width of a FLV file.
Media.totalTime	MediaDisplay, MediaPlayback	An integer that indicates the total length of the media, in seconds.
Media.volume	MediaDisplay, MediaPlayback	An integer from 0 (minimum) to 100 (maximum) that represents the volume level.

Event summary for the Media class

The following table lists events of the Media class.

Event	Components	Description
Media.change	MediaDisplay, MediaPlayback	Broadcast continuously while media is playing.
Media.click	MediaController, MediaPlayback	Broadcast when the user clicks the Play/Pause button.
Media.complete	MediaDisplay, MediaPlayback	Notification that the playhead has reached the end of the media.
Media.cuePoint	MediaDisplay, MediaPlayback	Notification that the playhead has reached a given cue point.
Media.playheadChange	MediaController, MediaPlayback	Broadcast by the component instance when a user moves the playback slider or clicks the Go to Beginning or Go to End button.
Media.progress	MediaDisplay, MediaPlayback	Generated continuously until the media has downloaded completely.
Media.volume	MediaController, MediaPlayback	Broadcast when the user adjusts the volume.

Media.activePlayControl

Applies to

MediaController

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia.activePlayControl

Description

Property; a string value that specifies the state the MediaController component should be in when it is loaded at runtime. A value of `"play"` indicates a play state; a value of `"pause"` indicates a paused state. Set this property and the `autoPlay` property such that both indicate the same state. The default value is `"play"`.

The button image displayed in the MediaController component is the opposite of the current play/pause state. For example, in the play state, the MediaController displays a pause button, because that is what would result from the user clicking the button and toggling the state.

Since it indicates the state that the controller will be in when it is loaded, the `activePlayControl` property must be set before the controller is created, either through the Property inspector or the Component inspector, if the component is on the Stage. If the component is being created by ActionScript code, this property must be set in the `initObj` parameter. Changing the value of this property after the component has been created has no effect. The value can be changed only by the user clicking the Play/Pause button.

Example

The following example indicates that the control will be paused when first loaded:

```
myMedia.activePlayControl = "pause";
```

See also

`Media.autoPlay`

Media.addCuePoint()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.addCuePoint(cuePointName, cuePointTime)
```

Parameters

`cuePointName` A string that names the cue point.

`cuePointTime` A number, in seconds, that indicates when a `cuePoint` event is broadcast.

Returns

Nothing.

Description

Method; adds a cue point object to a MediaPlayback or MediaDisplay instance. When the playhead time equals a cue point time, a `cuePoint` event is broadcast.

Example

The following code adds a cue point called `Homerun` to `myMedia` when the playhead time equals 16 seconds.

```
myMedia.addCuePoint("Homerun", 16);
```

See also

`Media.cuePoint`, `Media.cuePoints`, `Media.getCuePoint()`, `Media.removeAllCuePoints()`, `Media.removeCuePoint()`

Media.aspectRatio

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia`.aspectRatio`

Description

Property; a Boolean value that determines whether a MediaDisplay or MediaPlayback instance maintains its video aspect ratio during playback. A `true` value indicates that the aspect ratio should be maintained; a `false` value indicates that the aspect ratio can change during playback. The default value is `true`.

Example

The following example indicates that the aspect ratio can change during playback:

```
myMedia.aspectRatio = false;
```

Media.associateController()

Applies to

MediaDisplay

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.associateController(instanceName)
```

Parameters

instanceName A string that specifies the instance name of the MediaController component to associate.

Returns

Nothing.

Description

Method; associates a MediaDisplay instance with a MediaController instance.

If associate a MediaController instance with a MediaDisplay instance by using `Media.associateDisplay()`, you do not need to use `Media.associateController()`.

Example

The following code associates `myMedia` with `myController`:

```
myMedia.associateController(myController);
```

See also

`Media.associateDisplay()`

Media.associateDisplay()

Applies to

MediaController

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.associateDisplay(instanceName)
```

Parameters

instanceName A string that specifies the instance name of the MediaDisplay component to associate.

Returns

Nothing.

Description

Method; associates a MediaController instance with a MediaDisplay instance.

If you associate a MediaDisplay instance with a MediaController instance by using `Media.associateController()`, you do not need to use `Media.associateDisplay()`.

Example

The following code associates `myMedia` with `myDisplay`:

```
myMedia.associateDisplay(myDisplay);
```

See also

`Media.associateController()`

Media.autoPlay

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia`.autoPlay`

Description

Property; a Boolean value that determines whether the MediaPlayback or MediaDisplay instance should immediately start attempting to buffer and play. A `true` value indicates that the control buffers and plays at runtime; a `false` value indicates the control is stopped at runtime. This property depends on the `contentPath` and `mediaType` properties. If `contentPath` and `mediaType` are not set, no playback occurs at runtime. The default value is `true`.

Example

The following example indicates that the control is not started when first loaded:

```
myMedia.autoPlay = false;
```

See also

`Media.contentPath`, `Media.mediaType`

Media.autoSize

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia.autoSize

Description

Property; a Boolean value that determines the size of the media-viewing portion of the MediaDisplay or MediaPlayback component.

For the MediaDisplay component, the property behaves as follows:

- If you set this property to `true`, Flash displays the media at its preferred size, regardless of the size of the component. This implies that, unless the MediaDisplay instance size is the same as the size of the media, the media will either spill out of the instance boundaries or not fill the instance.
- If you set this property to `false`, Flash uses the instance size as much as possible, while honoring the aspect ratio. If both `Media.autoSize` and `Media.aspectRatio` are set to `false`, the exact size of the component is used.

For the MediaPlayback component, the property behaves as follows:

- If you set this property to `true`, Flash displays the media at its preferred size unless the media playback area is smaller than the preferred size. If this is the case, Flash shrinks the media to fit inside the instance and respect the aspect ratio. If the preferred size is smaller than the media area of the instance, part of the media area goes unused.
- If you set this property to `false`, Flash uses the instance size as much as possible, while honoring the aspect ratio. If both `Media.autoSize` and `Media.aspectRatio` are set to `false`, the media area of the component is filled. This area is defined as the area above the controls (in the default layout), minus a surrounding 8-pixel margin that makes up the edges of the component.

The default value is `true`.

Example

The following example indicates that the control is not played back according to its media size:

```
myMedia.autoSize = false;
```

See also

`Media.aspectRatio`

Media.backgroundStyle

Applies to

MediaController

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.backgroundStyle
```

Description

Property; a Boolean value that indicates which background is drawn for the MediaController instance. A value of `"default"` indicates that the chrome background is drawn, and a value of `"none"` indicates that no chrome background is drawn. The default value is `"default"`.

This is not a style property and therefore is not affected by style settings.

Example

The following example indicates that the chrome background will not be drawn for the control:

```
myMedia.backgroundStyle = "none";
```

Media.bytesLoaded

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.bytesLoaded
```

Description

Read-only property; the number of bytes already loaded into the component that are available for playing. The default value is `undefined`.

Example

The following code creates a variable called `PlaybackLoad` that will be set with the number of bytes loaded. The variable is then used in a `for` loop.

```
// create variable that holds how many bytes are loaded
var PlaybackLoad = myMedia.bytesLoaded;
```

```
// perform some function until playback ready
for (PlaybackLoad < 150) {
  someFunction();
}
```

Media.bytesTotal

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.bytesTotal
```

Description

Read-only property; the number of bytes to be loaded into the MediaPlayback or MediaDisplay component. The default value is `undefined`.

Example

The following example tells the user the size of the media to be streamed:

```
myTextField.text = myMedia.bytesTotal;
```

Media.change

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  // insert your code here
}
myMedia.addEventListener("change", listenerObject)
```

Description

Event; broadcast by the MediaDisplay and MediaPlayback components while the media is playing. The percentage complete can be retrieved from the component instance.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Media.change` event's event object has two additional properties:

`target` A reference to the broadcasting object.

`type` The string `"change"`, which indicates the type of event.

For more information, see "EventDispatcher class" on page 415.

Example

The following example uses an object listener to determine the playhead position (`Media.playheadTime`), from which the percentage complete can be calculated:

```
var myPlayerListener = new Object();
myPlayerListener.change = function(eventObject){
  var myPosition = myPlayer.playheadTime;
  var myPercentPosition = (myPosition/totalTime);
}
myPlayer.addEventListener("change", myPlayerListener);
```

See also

`Media.playing`, `Media.pause()`

Media.click

Applies to

MediaController, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.click = function(eventObject){
  // insert your code here
}
myMedia.addEventListener("click", listenerObject)
```

Description

Event; broadcast when the user clicks the Play/Pause button. The detail field can be used to determine which button was clicked. The `Media.click` event object has the following properties:

`detail` The string `"pause"` or `"play"`.

`target` A reference to the MediaController or MediaPlayback instance.

`type` The string `"click"`.

Example

The following example opens a pop-up window when the user clicks Play:

```
var myMediaListener = new Object()
myMediaListener.click = function(){
  PopUpManager.createPopup(_root, mx.containers.Window, false, {contentPath:
  movieSale});
}
myMedia.addEventListener("click", myMediaListener);
```

Media.complete

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.complete = function(eventObject){
  // insert your code here
}
myMedia.addEventListener("complete", listenerObject)
```

Description

Event; notification that the playhead has reached the end of the media. The `Media.complete` event object has the following properties:

`target` A reference to the MediaDisplay or MediaPlayback instance.

`type` The string `"complete"`.

Example

The following example uses an object listener to determine when the media has finished playing:

```
var myListener = new Object();
myListener.complete = function(eventObject) {
    trace("media is Finished");
};
myMedia.addEventListener("complete", myListener);
```

Media.contentPath

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.contentPath
```

Description

Property; a string that holds the relative path and filename of the media to be streamed and/or played. Setting the `contentPath` property is equivalent to calling the `Media.setMedia()` method without specifying a *`mediaType`* parameter. When no *`mediaType`* parameter is set with `Media.setMedia()`, the default type is FLV. The default value of the `contentPath` property is `undefined`.

Example

The following example displays the name of the media playing in a text box:

```
myTextField.text = myMedia.contentPath;
```

See also

`Media.setMedia()`

Media.controllerPolicy

Applies to

MediaController, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.controllerPolicy
```

Description

Property; determines whether the MediaController component (or the controller subcomponent within the MediaPlayback component) is hidden when instantiated and only appears when the user moves the mouse over the controller's collapsed state.

The possible values for this property are as follows:

- `"on"` specifies that the controls are always expanded.
- `"off"` specifies that the controls are always collapsed.
- `"auto"` (the default) specifies that the control remains in the collapsed state until the user moves the mouse over the hit area. The hit area matches the area in which the collapsed control is drawn. The control remains expanded until the mouse leaves the hit area.

Note: The hit area expands and contracts with the controller.

Example

The following example keeps the controller open at all times:

```
myMedia.controllerPolicy = "on";
```

Media.controlPlacement

Applies to

MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia`.controlPlacement`

Description

Property; determines where the controller portion of the MediaPlayback component is positioned in relation to its display. The possible values are `"top"`, `"bottom"`, `"left"`, and `"right"`. The default value is `"bottom"`.

Example

For the following example, the controller portion of the MediaPlayback component is on the right side:

```
myMedia.controlPlacement = "right";
```

Media.cuePoint

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.cuePoint = function(eventObject){
  // insert your code here
}
myMedia.addEventListener("cuePoint", listenerObject)
```

Description

Event; notification that the playhead has reached the cue point. The `Media.cuePoint` event object has the following properties:

`name` A string that indicates the name of the cue point.

`time` A number, expressed in frames or seconds, that indicates when the cue point was reached.

`target` A reference to the MediaPlayback object if there is one, or to the MediaDisplay object itself.

`type` The string `"cuePoint"`.

Example

The following example uses an object listener to determine when a cue point has been reached:

```
var myCuePointListener = new Object();
myCuePointListener.cuePoint = function(eventObject){
  trace("heard " + eventObject.type + ", " + eventObject.target);
}
myPlayback.addEventListener("cuePoint", myCuePointListener);
```

See also

`Media.addCuePoint()`, `Media.cuePoints`, `Media.getCuePoint()`

Media.cuePoints

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.cuePoints
```

or

```
myMedia.cuePoints[N]
```

Description

Property; an array of cue point objects that have been assigned to a MediaPlayback or MediaDisplay instance. In the array, each cue point object can have a name, a time in seconds or frames, and a player property (which is the instance name of the component it is associated with). The default value is an empty array (`[]`).

Example

The following example deletes the third cue point if playing an action preview:

```
if(myVariable == actionPreview) {
  myMedia.removeCuePoint(myMedia.cuePoints[2]);
}
```

See also

`Media.addCuePoint()`, `Media.getCuePoint()`, `Media.removeCuePoint()`

Media.displayFull()

Applies to

MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`myMedia``.displayFull()`

Parameters

None.

Returns

Nothing.

Description

Method; sets the MediaPlayback instance to full-screen mode. In this mode, the component expands to fill the entire Stage. To return the component to its normal size, use `Media.displayNormal()`.

Example

The following code forces the component to expand to fit the Stage:

```
myMedia.displayFull();
```

See also

`Media.displayNormal()`

Media.displayNormal()

Applies to

MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.displayNormal()
```

Parameters

None.

Returns

Nothing.

Description

Method; sets the MediaPlayback instance back to its normal size after a `Media.displayFull()` method has been used.

Example

The following code returns a MediaPlayback component to its original size:

```
myMedia.displayNormal();
```

See also

`Media.displayFull()`

Media.getCuePoint()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.getCuePoint(cuePointName)
```

Parameters

cuePointName The string that was provided when `Media.addCuePoint()` was used.

Returns

A cue point object.

Description

Method; returns a cue point object based on its cue point name.

Example

The following code retrieves a cue point named `myCuePointName`.

```
myMedia.removeCuePoint(myMedia.getCuePoint("myCuePointName"));
```

See also

`Media.addCuePoint()`, `Media.cuePoint`, `Media.cuePoints`, `Media.removeCuePoint()`

Media.horizontal

Applies to

MediaController

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`myMedia``.horizontal`

Description

Property; determines whether the MediaController component displays itself in a vertical or horizontal orientation. A `true` value indicates that the component is displayed in a horizontal orientation; a `false` value indicates a vertical orientation. When set to `false`, the playbar and playback slider move from bottom to top. The default value is `true`.

Example

The following example displays the MediaController component in a vertical orientation:

```
myMedia.horizontal = false;
```

Media.mediaType

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.mediaType
```

Description

Property; indicates the type of media (FLV or MP3) to be played. The default value is `"FLV"`. See "Importing Macromedia Flash Video (FLV) files" in *Using Flash*.

Example

The following example determines the current media type being played:

```
var currentMedia = myMedia.mediaType;
```

See also

`Media.setMedia()`

Media.pause()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.pause()
```

Parameters

None.

Returns

Nothing,

Description

Method; pauses the playhead at the current location.

Example

The following code pauses the playback.

```
myMedia.pause();
```

Media.play()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.play(startingPoint)
```

Parameters

startingPoint A non-negative integer that indicates the starting point (in seconds) at which the media should begin playing.

Returns

Nothing.

Description

Method; plays the media associated with the component instance at the given starting point. The default value is the current value of `playheadTime`.

Example

The following code indicates that the media component should start playing at 120 seconds:

```
myMedia.play(120);
```

See also

`Media.pause()`

Media.playheadChange

Applies to

MediaController, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.playheadChange = function(eventObject){
  // insert your code here
```

```
}
myMedia.addEventListener("playheadChange", listenerObject)
```

Description

Event; broadcast by the MediaController or MediaPlayback component when the user moves the playback slider or clicks the Go to Beginning or Go to End button. The `Media.playheadChange` event object has the following properties:

`detail` A number that indicates the percentage of the media that has played.

`type` The string `"playheadChange"`.

Example

The following example sends the percentage played to the Output panel when the user stops dragging the playhead:

```
var controlListen = new Object();
controlListen.playheadChange = function(eventObject){
  trace(eventObject.detail);
}
myMedia.addEventListener("playheadChange", controlListen);
```

Media.playheadTime

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.playheadTime
```

Description

Property; holds the current position of the playhead (in seconds) for the media Timeline that is playing. The default value is the location of the playhead.

Example

The following example sets a variable to the location of the playhead, which is indicated in seconds:

```
var myPlayhead = myMedia.playheadTime;
```

Media.playing

Applies to

MediaDisplay, MediaPlayback, MediaController

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia.playing

Description

Property; returns a Boolean value that indicates whether the media is playing (`true`) or paused (`false`). This property is read-only for the MediaDisplay and MediaPlayback components, and read/write for the MediaController component.

Example

The following code determines if the media is playing or paused:

```
if(myMedia.playing == true){
  some function;
}
```

See also

Media.change

Media.preferredHeight

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia.preferredHeight

Description

Property; set according to a FLV file's default height value. This property applies only to FLV media, because the height is fixed for MP3 files. This property can be used to set the height and width properties (plus some margin for the component itself). The default value is `undefined` if no FLV media is set.

Example

The following example sizes a MediaPlayback instance according to the media it is playing and accounts for the pixel margin needed for the component instance:

```
if(myPlayback.contentPath = !undefined){
  var mediaHeight = myPlayback.preferredHeight;
  var mediaWidth = myPlayback.preferredWidth;
  myPlayback.setSize((mediaWidth + 20), (mediaHeight + 70));
  }
```

Media.preferredWidth

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.preferredWidth
```

Description

Property; set according to a FLV file's default width value. The default value is `undefined`.

Example

The following example sets the desired width of the variable `mediaWidth`:

```
var mediaWidth = myMedia.preferredWidth;
```

Media.progress

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.progress = function(eventObject){
  // insert your code here
}
myMedia.addEventListener("progress", listenerObject)
```

Description

Event; is generated continuously until media has completely downloaded. The `Media.progress` event object has the following properties:

`target` A reference to the MediaDisplay or MediaPlayback instance.

`type` The string `"progress"`.

Example

The following example listens for progress:

```
var myProgressListener = new Object();
myProgressListener.progress = function(){
  // Make lightMovieClip blink while progress is occurring
  var lightVisible = lightMovieClip.visible;
  lightMovieClip.visible = !lightVisible;
}
```

The following example

```
// Duration of delay before calling timeOut
var timeOut:Number = 3000;

// if timeOut has been reached, do this:
function callback(arg) {
  trace(arg);
}

// Listen for progress
var myListener:Object = new Object();
myListener.progress = function(evt) {
  setInterval(callback, timeOut, "Experiencing Network Delay");
};
md.addEventListener("progress", myListener);
```

Media.removeAllCuePoints()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia`.removeAllCuePoints()`

Parameters

None.

Returns

Nothing.

Description

Method; deletes all cue point objects associated with a component instance.

Example

The following code deletes all cue point objects:

```
myMedia.removeAllCuePoints();
```

See also

Media.addCuePoint(), Media.cuePoints, Media.removeCuePoint()

Media.removeCuePoint()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

myMedia.removeCuePoint(*cuePoint*)

Parameters

cuePoint A reference to a cue point object that has been assigned previously by means of Media.addCuePoint().

Returns

Nothing.

Description

Method; deletes a cue point associated with a component instance.

Example

The following code deletes a cue point named myCuePoint:

```
myMedia.removeCuePoint(getCuePoint("myCuePoint"));
```

See also

Media.addCuePoint(), Media.cuePoints, Media.removeAllCuePoints()

Media.setMedia()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.setMedia(contentPath [, mediaType])
```

Parameters

contentPath A string that indicates the URL of the media to be played. The default value is `undefined`.

mediaType A string used to set the media type to either FLV or MP3. This parameter is optional. The default value is FLV.

Returns

Nothing.

Description

Method; sets the media type and path to the specified media type using a URL parameter.

This method provides the recommended way of setting the content path and media type for the MediaPlayback and MediaDisplay components. The `Media.contentPath` property can also be used to set the content path, but does not allow you to set the media type.

If you are working only with FLV files, you do not need to specify a *mediaType* parameter. If you are working exclusively with MP3 files, you must set the *mediaType* parameter to MP3 once. If you are switching back and forth between FLV and MP3 files, you must change the media type each time in your `setMedia()` call. If you attempt to play an MP3 file without explicitly setting the media type to MP3, the file will not play.

Example

The following code provides new media for a component instance to play.

```
myMedia.setMedia("http://www.RogerMoore.com/moonraker.flv", "FLV");
```

Media.stop()

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.stop()
```

Parameters

None.

Returns

Nothing.

Description

Method; stops the playhead and moves it to position 0, which is the beginning of the media.

Example

The following code stops the playhead and moves it to position 0:

```
myMedia.stop()
```

Media.totalTime

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.totalTime
```

Description

Property; the total length of the media, in seconds. Since the FLV file format does not provide its play time to a media component until it is completely loaded, you must input `Media.totalTime` manually so that the playbar can accurately reflect the actual play time of the media. The default value for MP3 files is the play time of the media. For FLV files, the default value is `undefined`.

You cannot set this property for MP3 files, because the information is contained in the Sound object.

Example

The following example sets the play time (in seconds) for the FLV media:

```
myMedia.totalTime = 151;
```

Media.volume

Applies to

MediaDisplay, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.volume
```

Description

Property; stores an integer that indicates the volume setting, which can range from 0 to 100. The default value is 75.

Example

The following example sets the maximum volume for media playback:

```
myMedia.volume = 100;
```

See also

Media.volume, Media.pause()

Media.volume

Applies to

MediaController, MediaPlayback

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.volume = function(eventObject){
  // insert your code here
}
myMedia.addEventListener("volume", listenerObject)
```

Description

Event; broadcast when the volume value is adjusted by the user. The `Media.volume` event object has the following properties:

`detail` An integer between 0 and 100 that represents the volume level.

`type` The string `"volume"`.

Example

The following example informs the user that the volume is being adjusted:

```
var myVolListener = new Object();
myVolListener.volume = function(){
   mytextfield.text = "Volume adjusted!";
}
myMedia.addEventListener("volume", myVolListener);
```

See also

`Media.volume`

Menu component (Flash Professional only)

The Menu component lets a user select an item from a pop-up menu, much like the File or Edit menu of most software applications.

A Menu component usually opens in an application when a user rolls over or clicks a button-like menu activator. You can also script a Menu component to open when a user presses a certain key.

Menu components are always created dynamically at runtime. You must add the component to the document from the Components panel, and delete it to add it to the library. Then, use the following code to create a menu with ActionScript:

```
var myMenu = mx.controls.Menu.createMenu(parent, menuDataProvider);
```

Use the following code to open a menu in an application:

```
myMenu.show(x, y);
```

A `menuShow` event is broadcast to all of the Menu instance's listeners immediately before the menu is rendered, so you can update the state of the menu items. Similarly, immediately after a Menu instance is hidden, a `menuHide` event is broadcast.

The items in a menu are described by XML. For more information, see "Understanding the Menu component: view and data" on page 540.

You cannot make the Menu component accessible to screen readers.

Interacting with the Menu component (Flash Professional only)

You can use the mouse and keyboard to interact with a Menu component.

After a Menu component is opened, it remains visible until it is closed by a script or until the user clicks the mouse outside the menu or inside an enabled item.

Clicking selects a menu item, except with the following types of menu items:

Disabled items or separators Rollovers and clicks have no effect (the menu remains visible).

Anchors for a submenu Rollovers activate the submenu; clicks have no effect; rolling onto any item other than those of the submenu closes the submenu.

When an item is selected, a `Menu.change` event is sent to all of the menu's listeners, the menu is hidden, and the following actions occur, depending on item type:

check The item's `selected` attribute is toggled.

radio The item becomes the current selection of its radio group.

Moving the mouse triggers `Menu.rollOut` and `Menu.rollOver` events.

Pressing the mouse outside the menu closes the menu and triggers a `Menu.menuHide` event.

Releasing the mouse in an enabled item affects item types in the following ways:

check The item's `selected` attribute is toggled.

radio The item's `selected` attribute is set to `true`, and the previously selected item's `selected` attribute in the radio group is set to `false`. The `selection` property of the corresponding radio group object is set to refer to the selected menu item.

undefined and the parent of a hierarchical menu The visibility of the hierarchical menu is toggled.

When a Menu instance has focus either from clicking or tabbing, you can use the following keys to control it:

Key	Description
Down Arrow Up Arrow	Moves the selection down and up the rows of the menu. The selection cycles at the top or bottom row.
Right Arrow	Opens a submenu, or moves selection to the next menu in a menu bar (if a menu bar exists).
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. If a submenu does not exist, this key has the same effect as clicking and releasing on a row.

Note: If a menu is opened, you can press the Tab key to move out of the menu. You must either make a selection or dismiss the menu by pressing Escape.

Using the Menu component (Flash Professional only)

You can use the Menu component to create a menu of selectable choices; this menu is like the File or Edit menu of most software applications. You can also use the Menu component to create context-sensitive menus that appear when a user clicks a hot spot or a presses a modifier key. Use the Menu component with the MenuBar component to create a horizontal menu bar with menus that extend under each menu bar item.

Like standard desktop menus, the Menu component supports menu items whose functions fall into the following general categories:

Command activators These items trigger events; you write code to handle those events.

Submenu anchors These items are anchors that open submenus.

Radio buttons These items operate in groups; you can select only one item at a time.

Check box items These items represent a Boolean (`true` or `false`) value.

Separators These items provide a simple horizontal line that divides the items in a menu into different visual groups.

Understanding the Menu component: view and data

Conceptually, the Menu component consists of a data model and a view that displays the data. The Menu class provides the view and contains the visual configuration methods. The MenuDataProvider class adds methods to the global XML prototype object (much like the DataProvider API does to the Array object); these methods let you externally construct data providers and add them to multiple menu instances. The data provider broadcasts any changes to all of its client views. (See "MenuDataProvider class" on page 568.)

A Menu instance is a hierarchical collection of XML elements that correspond to individual menu items. The attributes define the behavior and appearance of the corresponding menu item on the screen. The collection is easily translated to and from XML, which is used to describe menus (the `menu` tag) and items (the `menuitem` tag). The built-in ActionScript XML class is the basis for the model underlying the Menu component.

A simple menu with two items can be described in XML with two menu item subelements:

```
<menu>
  <menuitem label="Up" />
  <menuitem label="Down" />
</menu>
```

Note: The tag names of the XML nodes (`menu` and `menuitem`) are not important; the attributes and their nesting relationships are used in the menu.

About hierarchical menus

To create hierarchical menus, embed XML elements within a parent XML element, as follows:

```
<menu>
    <menuitem label="MenuItem A" >
      <menuitem label="SubMenuItem 1-A" />
      <menuitem label="SubMenuItem 2-A" />
    </menuitem>
    <menuitem label="MenuItem B" >
      <menuitem label="SubMenuItem 1-B" />
      <menuitem label="SubMenuItem 2-B" />
    </menuitem>
</menu>
```

This converts the parent menu item into a pop-up menu anchor, so it does not generate events when selected.

About menu item XML attributes

The attributes of a menu item XML element determine what is displayed, how the menu item behaves, and how it is exposed to ActionScript. The following table describes the attributes of an XML menu item:

Attribute name	Type	Default	Description
`label`	String	`undefined`	The text that is displayed to represent a menu item. This attribute is required for all item types, except `separator`.
`type`	`separator`, `check`, `radio`, `normal`, or `undefined`	`undefined`	The type of menu item: `separator`, `check box`, `radio button`, or `normal` (a command or submenu activator). If this attribute does not exist, the default value is `normal`.
`icon`	String	`undefined`	The linkage identifier of an image asset. This attribute is not required and is not available for the `check`, `radio`, or `separator` type.
`instanceName`	String	`undefined`	An identifier that you can use to reference the menu item instance from the root menu instance. For example, a menu item named *yellow* can be referenced as `myMenu.yellow`. This attribute is not required.
`groupName`	String	`undefined`	An identifier that you can 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. For example, a radio group named *colors* can be referenced as `myMenu.colors`. This attribute is required only for the type `radio`.
`selected`	A Boolean value (`false` or `true`) or string (`"false"` or `"true"`)	`false`	A Boolean or string value indicating whether a `check` or `radio` item is on (`true`) or off (`false`). This attribute is not required.
`enabled`	A Boolean value (`false` or `true`) or string (`"false"` or `"true"`)	`true`	A Boolean or string value indicating whether this menu item can be selected (`true`) or not (`false`). This attribute is not required.

About menu item types (Flash Professional only)

There are four kinds of menu items, specified by the `type` attribute:

```
<menu>
    <menuitem label="Normal Item" />
    <menuitem type="separator" />
    <menuitem label="Checkbox Item" type="check" instanceName="check_1"/>
    <menuitem label="RadioButton Item" type="radio" groupName="radioGroup_1" /
  >
</menu>
```

Normal menu items

The `Normal Item` menu item doesn't have a `type` attribute, which means that the `type` attribute defaults to `normal`. Normal items can be command activators or submenu activators, depending on whether they have nested subitems.

Separator menu items

A menu item whose `type` attribute is set to `separator` acts as a visual divider in a menu. The following XML creates three menu items, Top, Middle, and Bottom, with separators between them:

```
<menu>
    <menuitem label="Top" />
    <menuitem type="separator" />
    <menuitem label="Middle" />
    <menuitem type="separator" />
    <menuitem label="Bottom" />
</menu>
```

All separator items are disabled. Clicking on or rolling over a separator has no effect.

Check box menu items

A menu item whose `type` attribute is set to `check` acts as check box item in the menu; when the `selected` attribute is set to `true`, a check mark appears beside the menu item's label. When a check box item is selected, its state automatically toggles, and a `change` event is broadcast to all listeners on the root menu. The following example defines three check box menu items:

```
<menu>
    <menuitem label="Apples" type="check" instanceName="buyApples"
  selected="true" />
    <menuitem label="Oranges" type="check" instanceName="buyOranges"
  selected="false" />
    <menuitem label="Bananas" type="check" instanceName="buyBananas"
  selected="false" />
</menu>
```

You can use the instance names in ActionScript to access the menu items directly from the menu itself, as in the following example:

```
myMenu.setMenuItemSelected(myMenu.buyapples, true);
myMenu.setMenuItemSelected(myMenu.buyoranges, false);
```

Note: The `selected` attribute should be modified only with the `setMenuItemSelected()` method. You can directly examine the `selected` attribute, but it returns a string value of `true` or `false`.

Radio button menu items

Menu items whose `type` attribute is set to `radio` can be grouped together so that only one of the items can be selected at a time. You create a radio group by giving the menu items the same value for their `groupName` attribute, as in the following example:

```
<menu>
    <menuitem label="Center" type="radio" groupName="alignment_group"
       instanceName="center_item"/>
    <menuitem type="separator" />
    <menuitem label="Top"    type="radio" groupName="alignment_group" />
    <menuitem label="Bottom" type="radio" groupName="alignment_group" />
    <menuitem label="Right"  type="radio" groupName="alignment_group" />
    <menuitem label="Left"   type="radio" groupName="alignment_group" />
</menu>
```

When the user selects one of the items, the current selection automatically changes, and a `change` event is broadcast to all listeners on the root menu. The currently selected item in a radio group is available in ActionScript through the `selection` property, as follows:

```
var selectedMenuItem = myMenu.alignment_group.selection;
myMenu.alignment_group = myMenu.center_item;
```

Each `groupName` value must be unique within the scope of the root menu instance.

Note: The `selected` attribute should be modified only with the `setMenuItemSelected()` method. You can directly examine the `selected` attribute, but it returns a string value of `true` or `false`.

Exposing menu items to ActionScript

You can assign each menu item a unique identifier in the `instanceName` attribute, which makes the menu item accessible directly from the root menu. For example, the following XML code provides `instanceName` attributes for each menu item:

```
<menu>
    <menuitem label="Item 1" instanceName="item_1" />
    <menuitem label="Item 2" instanceName="item_2" >
      <menuitem label="SubItem A" instanceName="sub_item_A" />
      <menuitem label="SubItem B" instanceName="sub_item_B" />
    </menuitem>
</menu>
```

You can use ActionScript to access the corresponding instances and their attributes directly from the menu component, as follows:

```
var aMenuItem = myMenu.item_1;
myMenu.setMenuItemEnabled(item_2, true);
var aLabel = myMenu.sub_item_A.label;
```

Note: Each `instanceName` attribute must be unique within the scope of the root menu component instance (including all of the submenus of root).

About initialization object properties (Flash Professional only)

The *initObject* (initialization object) parameter is a fundamental concept in creating the layout for the Menu component. This parameter is an object with properties. Each property represents one of the possible the XML attributes of a menu item. (For a description of the properties allowed in the *initObject* parameter, see "About menu item XML attributes" on page 541.)

The *initObject* parameter is used in the following methods:

- `Menu.addMenuItem()`
- `Menu.addMenuItemAt()`
- `MenuDataProvider.addMenuItem()`
- `MenuDataProvider.addMenuItemAt()`

The following example creates an *initObject* parameter with two properties, `label` and `instanceName`:

```
var i = myMenu.addMenuItem({label:"myMenuItem", instanceName:"myFirstItem"});
```

Several of the properties work together to create a particular type of menu item. You assign specific properties to create certain types of menu items (normal, separator, check box, or radio button).

For example, you can initialize a normal menu item with the following *initObject* parameter:

```
myMenu.addMenuItem({label:"myMenuItem", enabled:true, icon:"myIcon",
  instanceName:"myFirstItem"});
```

You can initialize a separator menu item with the following *initObject* parameter:

```
myMenu.addMenuItem({type:"separator"});
```

You can initialize a check box menu item with the following *initObject* parameter:

```
myMenu.addMenuItem({type:"check", label:"myMenuCheck", enabled:false,
  selected:true, instanceName:"myFirstCheckItem"})
```

You can initialize a radio button menu item with the following *initObject* parameter:

```
myMenu.addMenuItem({type:"radio", label:"myMenuRadio1", enabled:true,
  selected:false, groupName:"myRadioGroup" instanceName:"myFirstRadioItem"})
```

You should treat the `instanceName`, `groupName`, and `type` attributes of a menu item as read-only. You should set them only while creating an item (for example, in a call to `addMenuItem()`). Modifying these attributes after creation may produce unpredictable results.

Menu parameters (Flash Professional only)

There are no authoring parameters for the Menu component.

You can write ActionScript to control the Menu component using its properties, methods, and events. For more information, see "Menu class (Flash Professional only)" on page 551.

Creating an application with the Menu component (Flash Professional only)

In the following example, a developer is building an application and uses the Menu component to expose some of the commands that users can issue, such as Open, Close, and Save.

To create an application with the Menu component:

1. Select File > New and create a Flash document.
2. Drag the Menu component from the Components panel to the Stage and delete it.

   This adds the Menu component to the library without adding it to the application. Menus are created dynamically through ActionScript.
3. Drag a Button component from the Components panel to the Stage.

   The button will be used to activate the menu.
4. In the Property inspector, give the button the instance name **commandBtn**, and change its text property to **Commands**.
5. In the Actions panel on the first frame, enter the following code to add an event listener to listen for `click` events on the `commandBtn` instance:

```
// Create a menu
var myMenu = mx.controls.Menu.createMenu();

// Add some menu items
myMenu.addMenuItem("Open");
myMenu.addMenuItem("Close");
myMenu.addMenuItem("Save");
myMenu.addMenuItem("Revert");

// Add a change-listener to Menu to detect which menu item is selected
var changeListener = new Object();
changeListener.change = function(event) {
  var item = event.menuItem;
  trace("Item selected: " + item.attributes.label);
}
myMenu.addEventListener("change", changeListener);

// Add a button that displays the menu when the button is clicked
var listener = new Object();
listener.click = function(evtObj) {
  var button = evtObj.target;// get reference to the button

  // Display the menu at the bottom of the button
  _root.myMenu.show(button.x, button.y + button.height);
}
commandBtn.addEventListener("click", listener);
```

6. Select Control > Test Movie.

   Click the Commands button to see the menu appear. When you select a menu item, a `trace()` statement reports the selection in the Output panel.

To use XML data from a server to create and populate a menu:

1. Select File > New and create a Flash document.
2. Drag the Menu component from the Components panel to the Stage and delete it.

   This adds the Menu component to the library without adding it to the application. Menus are created dynamically through ActionScript.
3. In the Actions panel, add the following code to the first frame to create a menu and add some items:

```
var myMenu = mx.controls.Menu.createMenu();
// Import an XML file
var loader = new XML();
loader.menu = myMenu;
loader.ignoreWhite = true;
loader.onLoad = function(success) {
  // When the data arrives, pass it to the menu
  if(success) {
    this.menu.dataProvider = this.firstChild;
  }
};
loader.load(url);
```

   Note: The menu items are described by the children of the XML document's first child.

4. Select Control > Test Movie.

To use a well-formed XML string to create and populate a menu:

1. Select File > New and create a Flash document.
2. Drag the Menu component from the Components panel to the Stage and delete it.

   This adds the Menu component to the library without adding it to the application. Menus are created dynamically through ActionScript.
3. In the Actions panel, add the following code to the first frame to create a menu and add some items:

```
// Create an XML string containing a menu definition
var s = "";
s += "<menu>";
s += "<menuitem label='Undo' />";
s += "<menuitem type='separator' />";
s += "<menuitem label='Cut' />";
s += "<menuitem label='Copy' />";
s += "<menuitem label='Paste' />";
s += "<menuitem label='Clear' />";
s += "<menuitem type='separator' />";
s += "<menuitem label='Select All' />";
s += "</menu>";
// Create an XML object from the string
var xml = new XML(s);
xml.ignoreWhite = true;
// Create a menu from the XML object's first child
var myMenu = mx.controls.Menu.createMenu(_root, xml.firstChild);
```

4. Select Control > Test Movie.

To use the MenuDataProvider class to create and populate a menu:

1. Select File > New and create a Flash document.
2. Drag the Menu component from the Components panel to the Stage and delete it.

   This adds the Menu component to the library without adding it to the application. Menus are created dynamically through ActionScript.
3. In the Actions panel, add the following code to the first frame to create a menu and add some items:

```
// Create an XML object to act as a factory.
var xml = new XML();

// The item created next will not appear in the menu.
// The createMenu() method call (below) expects to
// receive a root element whose children will become
// the items. This is just a simple way to create that
// root element and give it a convenient name along
// the way.
var theMenuElement = xml.addMenuItem("Edit");

// Add the menu items.
theMenuElement.addMenuItem({label:"Undo"});
theMenuElement.addMenuItem({type:"separator"});
theMenuElement.addMenuItem({label:"Cut"});
theMenuElement.addMenuItem({label:"Copy"});
theMenuElement.addMenuItem({label:"Paste"});
theMenuElement.addMenuItem({label:"Clear", enabled:"false"});
theMenuElement.addMenuItem({type:"separator"});
theMenuElement.addMenuItem({label:"Select All"});
// Create the Menu object.
var theMenuControl = mx.controls.Menu.createMenu(_root, theMenuElement);
```

4. Select Control > Test Movie.

Customizing the Menu component (Flash Professional only)

The menu sizes itself horizontally to fit its widest text. You can also call the `setSize()` method to size the component. Icons should be sized to a maximum of 16 by 16 pixels.

Using styles with the Menu component

You can call the `setStyle()` method to change the style of the menu, its items, and its submenus.The Menu component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`alternatingRowColors`	Both	Specifies colors for rows in an alternating pattern. The value can be an array of two or more colors, for example, 0xFF00FF, 0xCC6699, and 0x996699. Unlike single-value color styles, `alternatingRowColors` does not accept color names; the values must be numeric color codes. By default, this style is not set, and `backgroundColor` is used in its place for all rows.
`backgroundColor`	Both	The background color of the menu. The default color is white and is defined on the class style declaration. This style is ignored if `alternatingRowColors` is specified.
`backgroundDisabledColor`	Both	The background color when the component's `enabled` property is set to `"false"`. The default value is 0xDDDDDD (medium gray).
`border styles`	Both	The Menu component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style is `"menuBorder"`.
`color`	Both	The text color.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textAlign`	Both	The text alignment: either `"left"`, `"right"`, or `"center"`. The default value is `"left"`.

Style	Theme	Description
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".
textIndent	Both	A number indicating the text indent. The default value is 0.
defaultIcon	Both	The name of the default icon to display on each row. The default value is undefined, which means no icon is displayed.
popupDuration	Both	The duration of the transition as a menu opens. The value is specified in milliseconds; 0 indicates no transition. The default value is 150.
rollOverColor	Both	The background color of a rolled-over row. The default value is 0xE3FFD6 (bright green) with the Halo theme and 0xAAAAAA (light gray) with the Sample theme. When themeColor is changed through a setStyle() call, the framework sets rollOverColor to a value related to the themeColor chosen.
selectionColor	Both	The background color of a selected row. The default value is a 0xCDFFC1 (light green) with the Halo theme and 0xEEEEEE (very light gray) with the Sample theme. When themeColor is changed through a setStyle() call, the framework sets selectionColor to a value related to the themeColor chosen.
selectionDuration	Both	The length of the transition from a normal to selected state, in milliseconds. The default value is 200.
selectionEasing	Both	A reference to the easing equation used to control the transition between selection states. The default equation uses a sine in/out formula. For more information, see "Customizing component animations" on page 75.
textRollOverColor	Both	The color of text when the mouse pointer rolls over. The default value is 0x2B333C (dark gray). This style is important when you set rollOverColor, because the two settings must complement each other so that text is easily viewable during rollover.
textSelectedColor	Both	The color of text in the selected row. The default value is 0x005F33 (dark gray). This style is important when you set selectionColor, because the two must complement each other so that text is easily viewable while selected.
useRollOver	Both	Determines whether rolling over a row activates highlighting. The default value is true.

Setting styles for all Menu components in a document

The Menu class inherits from the ScrollSelectList class. The default class-level style properties are defined on the ScrollSelectList class, which is shared by all List-based components. You can set new default style values on this class directly, and the new settings will be reflected in all affected components.

```
_global.styles.ScrollSelectList.setStyle("backgroundColor", 0xFF00AA);
```

To set a style property on the Menu components only, you can create a new `CSSStyleDeclaration` and store it in `_global.styles.Menu`.

```
import mx.styles.CSSStyleDeclaration;
if (_global.styles.Menu == undefined) {
  _global.styles.Menu = new CSSStyleDeclaration();
}
_global.styles.Menu.setStyle("backgroundColor", 0xFF00AA);
```

When creating a new class-level style declaration, you will lose all default values provided by the `ScrollSelectList` declaration. This includes `backgroundColor`, which is required for supporting mouse events. To create a class-level style declaration and preserve defaults, use a `for..in` loop to copy the old settings to the new declaration.

```
var source = _global.styles.ScrollSelectList;
var target = _global.styles.Menu;
for (var style in source) {
  target.setStyle(style, source.getStyle(style));
}
```

For more information about class-level styles see "Setting styles for a component class" on page 71.

Using skins with the Menu component

The Menu component uses an instance of RectBorder for its border (seesee "RectBorder class" on page 647).

The Menu component has visual assets for the branch, check mark, radio dot, and separator graphics. These assets are not dynamically skinnable, but the assets can be copied from the Flash UI Components 2/Themes/MMDefault/Menu Assets/States folder in both themes and modified as desired. The linkage identifiers cannot be changed, and all Menu instances must use the same symbols.

To create movie clip symbols for Menu assets:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the Menu Assets folder to the library for your document.
4. Expand the Menu Assets/States folder in the library of your document.

5. Open the symbols you want to customize for editing.

   For example, open the MenuCheckEnabled symbol.

6. Customize the symbol as desired.

   For example, change the image to be an X instead of a check mark.

7. Repeat steps 6-7 for all symbols you want to customize.
8. Click the Back button to return to the main Timeline.
9. Drag a Menu component to the Stage and delete it.

   This adds the Menu component to the library and makes it available at runtime.

10. Add ActionScript to the main timline to create a Menu instance at runtime:

```
var myMenu = mx.controls.Menu.createMenu();
myMenu.addMenuItem({label: "One", type: "check", selected: true});
myMenu.addMenuItem({label: "Two", type: "check"});
myMenu.addMenuItem({label: "Three", type: "check"});
myMenu.show(0, 0);
```

11. Select Control > Test Movie.

Menu class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > ScrollSelectList > Menu

ActionScript Class Name mx.controls.Menu

The methods and properties of the Menu class let you create and edit menus at runtime.

Setting a property of the menu class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.Menu.version);
```

Note: The code `trace(myMenuInstance.version);` returns `undefined`.

Method summary for the Menu class

The following table lists methods of the Menu class.

Method	Description
`Menu.addMenuItem()`	Adds a menu item to the menu.
`Menu.addMenuItemAt()`	Adds a menu item to the menu at a specific location.
`Menu.createMenu()`	Creates an instance of the Menu class. This is a static method.
`Menu.getMenuItemAt()`	Gets a reference to a menu item at a specified location.
`Menu.hide()`	Closes a menu.

Method	Description
`Menu.indexOf()`	Returns the index of a given menu item.
`Menu.removeAll()`	Removes all items from a menu.
`Menu.removeMenuItem()`	Removes the specified menu item.
`Menu.removeMenuItemAt()`	Removes a menu item from a menu at a specified location.
`Menu.setMenuItemEnabled()`	Indicates whether a menu item is enabled (`true`) or not (`false`).
`Menu.setMenuItemSelected()`	Indicates whether a menu item is selected (`true`) or not (`false`).
`Menu.show()`	Opens a menu at a specific location or at its previous location.

Methods inherited from the UIObject class

The following table lists the methods the Menu class inherits from the UIObject class. When calling these methods from the Menu object, use the form *`MenuInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Menu class inherits from the UIComponent class. When calling these methods from the Menu object, use the form *`MenuInstance.methodName`*.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Property summary for the Menu class

The following table lists the property of the Menu class.

Property	Description
`Menu.dataProvider`	The data source for a menu.

Properties inherited from the UIObject class

The following table lists the properties the Menu class inherits from the UIObject class. When accessing these properties from the Menu object, use the form *`MenuInstance.propertyName`*.

Property	Description
`UIObject.bottom`	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
`UIObject.height`	The height of the object, in pixels. Read-only.
`UIObject.left`	The left edge of the object, in pixels. Read-only.
`UIObject.right`	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
`UIObject.scaleX`	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
`UIObject.scaleY`	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
`UIObject.top`	The position of the top edge of the object, relative to its parent. Read-only.
`UIObject.visible`	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
`UIObject.width`	The width of the object, in pixels. Read-only.
`UIObject.x`	The left edge of the object, in pixels. Read-only.
`UIObject.y`	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Menu class inherits from the UIComponent class. When accessing these properties from the Menu object, use the form *`MenuInstance.propertyName`*.

Property	Description
`UIComponent.enabled`	Indicates whether the component can receive focus and input.
`UIComponent.tabIndex`	A number indicating the tab order for a component in a document.

Event summary for the Menu class

The following table lists events of the Menu class.

Event	Description
`Menu.change`	Broadcast when a user causes a change in a menu.
`Menu.menuHide`	Broadcast when a menu closes.
`Menu.menuShow`	Broadcast when a menu opens.
`Menu.rollOut`	Broadcast when the pointer rolls off an item.
`Menu.rollOver`	Broadcast when the pointer rolls over an item.

Events inherited from the UIObject class

The following table lists the events the Menu class inherits from the UIObject class.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Menu class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

Menu.addMenuItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenu.addMenuItem(initObject)
```

Usage 2:

```
myMenu.addMenuItem(childMenuItem)
```

Parameters

initObject An object containing properties that initialize a menu item's attributes. See "About menu item XML attributes" on page 541.

childMenuItem An XML node object.

Returns

A reference to the added XML node.

Description

Method; Usage 1 adds a menu item at the end of the menu. The menu item is constructed from the values supplied in the *initObject* parameter. Usage 2 adds a menu item that is a prebuilt XML node (in the form of an XML object) at the end of the menu. Adding a preexisting node removes the node from its previous location.

Example

Usage 1: The following example appends a menu item to a menu:

```
myMenu.addMenuItem({label:"Item 1", type:"radio", selected:false,
  enabled:true, instanceName:"radioItem1", groupName:"myRadioGroup"});
```

Usage 2: The following example moves a node from one menu to the root of another menu:

```
myMenu.addMenuItem(mySecondMenu.getMenuItemAt(3));
```

Menu.addMenuItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenu.addMenuItemAt(index, initObject)
```

Usage 2:

```
myMenu.addMenuItemAt(index, childMenuItem)
```

Parameters

index An integer indicating the index position (among the child nodes) at which the item is added.

initObject An object containing properties that initialize a menu item's attributes. See "About menu item XML attributes" on page 541.

childMenuItem An XML node object.

Returns

A reference to the added XML node.

Description

Method; Usage 1 adds a menu item (child node) at the specified location in the menu. The menu item is constructed from the values supplied in the *initObject* parameter. Usage 2 adds a menu item that is a prebuilt XML node (in the form of an XML object) at a specified location in the menu. Adding a preexisting node removes the node from its previous location.

Example

Usage 1: The following example adds a new node as the second child of the root of the menu:

```
myMenu.addMenuItemAt(1, {label:"Item 1", instanceName:"radioItem1",
  type:"radio", selected:false, enabled:true, groupName:"myRadioGroup"});
```

Usage 2: The following example moves a node from one menu to the fourth child of the root of another menu:

```
myMenu.addMenuItemAt(3, mySecondMenu.getMenuItemAt(3));
```

Menu.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  // insert your code here
}
myMenu.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners whenever a user causes a change in the menu.

Version 2 components use a dispatcher-listener event model. When a Menu component broadcasts a `change` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*listenerObject*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Menu.change` event's event object has the following additional properties:

- `menuBar` A reference to the MenuBar instance that is the parent of the target menu. When the target menu does not belong to a MenuBar instance, this value is `undefined`.
- `menu` A reference to the Menu instance where the target item is located.
- `menuItem` An XML node that is the menu item that was selected.
- `groupName` A string indicating the name of the radio button group to which the item belongs. If the item is not in a radio button group, this value is `undefined`.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `listener` is defined and passed to `myMenu.addEventListener()` as the second parameter. The event object is captured by the `change` handler in the `evt` parameter. When the `change` event is broadcast, a `trace` statement is sent to the Output panel.

```
listener = new Object();
listener.change = function(evt){
    trace("Menu item chosen: "+evt.menuItem.attributes.label);
}
myMenu.addEventListener("change", listener);
```

Menu.createMenu()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
Menu.createMenu([parent [, mdp]])
```

Parameters

parent A MovieClip instance. The movie clip is the parent component that contains the new Menu instance. This parameter is optional.

mdp The MenuDataProvider instance that describes this Menu instance. This parameter is optional.

Returns

A reference to the new menu instance.

Description

Method (static); instantiates a Menu instance, and optionally attaches it to the specified parent, with the specified MenuDataProvider as the data source for the menu items.

If the *parent* parameter is omitted or null, the Menu is attached to the `_root` Timeline.

If the *mdp* parameter is omitted or null, the menu does not have any items; you must call `addMenu()` or `setDataProvider()` to populate the menu.

Example

In the following example, line 1 creates a MenuDataProvider instance, which is an XML object that has the methods of the MenuDataProvider class.

In the following example, the first line creates an XML object instance, which is given the methods of the MenuDataProvider class (because the MenuDataProvider class is a decorator class of the XMLNode class). The next line adds a menu item (New) with a submenu (File, Project, and Resource). The next block of code adds more items to the main menu. The third block of code creates an empty menu attached to `myParentClip`, fills it with the data source `myMDP`, and opens it at the coordinates (100,20):

```
var myMDP:XML = new XML();
var newItem:Object = myMDP.addMenuItem({label:"New"});
  newItem.addMenuItem({label:"File..."});
  newItem.addMenuItem({label:"Project..."});
  newItem.addMenuItem({label:"Resource..."});

myMDP.addMenuItem({label:"Open", instanceName:"miOpen"});
myMDP.addMenuItem({label:"Save", instanceName:"miSave"});
myMDP.addMenuItem({type:"separator"});
myMDP.addMenuItem({label:"Quit", instanceName:"miQuit"});

var myMenu:mx.controls.Menu = mx.controls.Menu.createMenu(myParentClip,
  myMDP);
myMenu.show(100, 20);
```

To test this code, place it in the Actions panel on Frame 1 of the main Timeline. Drag a Menu component from the Components panel to the Stage and delete it. (This adds it to the library without placing it in the document.)

Menu.dataProvider

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenu`.dataProvider`

Description

Property; the data source for items in a Menu component.

`Menu.dataProvider` is an XML node object. Setting this property replaces the existing data source of the menu.

The default value is `undefined`.

Note: All XML or XMLNode instances are automatically given the methods and properties of the MenuDataProvider class when they are used with the Menu component.

Example

The following example imports an XML file and assigns it to the `Menu.dataProvider` property:

```
var myMenuDP = new XML();
myMenuDP.load("http://myServer.myDomain.com/source.xml");
myMenuDP.onLoad = function(){
  myMenuControl.dataProvider = myMenuDP;
}
```

Menu.getMenuItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenu.getMenuItemAt(*index*)

Parameters

index An integer indicating the index of the node in the menu.

Returns

A reference to the specified node.

Description

Method; returns a reference to the specified child node of the menu.

Example

The following example gets a reference to the second child node in `myMenu` and copies the value into the variable `myItem`:

```
var myItem = myMenu.getMenuItemAt(1);
```

Menu.hide()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenu.hide()
```

Parameters

None.

Returns

Nothing.

Description

Method; closes a menu.

Example

The following example retracts an extended menu:

```
myMenu.hide();
```

See also

Menu.show()

Menu.indexOf()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenu.indexOf(item)
```

Parameters

item A reference to an XML node that describes a menu item.

Returns

The index of the specified menu item, or `undefined` if the item does not belong to this menu.

Description

Method; returns the index of the specified menu item within this menu instance.

Example

The following example adds a menu item to a parent item and then gets the item's index within its parent:

```
var myItem = myMenu.addMenuItem({label:"That item"});
var myIndex = myMenu.indexOf(myItem);
```

Menu.menuHide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.menuHide = function(eventObject){
  // insert your code here
}
myMenu.addEventListener("menuHide", listenerObject)
```

Description

Event; broadcast to all registered listeners whenever a menu closes.

Version 2 components use a dispatcher-listener event model. When a Menu component dispatches a `menuHide` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (`listenerObject`) that you create. You call the `addEventListener()` method and pass it the name of the handler and the name of the listener object as parameters.

When the event is triggered, it automatically passes an event object (`eventObject`) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Menu.menuHide` event's event object has two additional properties:

- `menuBar` A reference to the MenuBar instance that is the parent of the target menu. When the target menu does not belong to a MenuBar instance, this value is `undefined`.
- `menu` A reference to the Menu instance that is hidden.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `form` is defined and passed to `myMenu.addEventListener()` as the second parameter. The event object is captured by the `menuHide` handler in the `evt` parameter. When the `menuHide` event is broadcast, a `trace` statement is sent to the Output panel.

```
form = new Object();
form.menuHide = function(evt){
    trace("Menu closed: "+evt.menu);
```

```
}
myMenu.addEventListener("menuHide", form);
```

See also

Menu.menuShow

Menu.menuShow

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.menuShow = function(eventObject){
  // insert your code here
}
myMenu.addEventListener("menuShow", listenerObject)
```

Description

Event; broadcast to all registered listeners whenever a menu opens. All parent nodes open menus to show their children.

Version 2 components use a dispatcher-listener event model. When a Menu component dispatches a `menuShow` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (`listenerObject`) that you create. You call the `addEventListener()` method and pass it the name of the handler and listener object as parameters.

When the event is triggered, it automatically passes an event object (`eventObject`) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Menu.menuShow` event's event object has two additional properties:

- `menuBar` A reference to the MenuBar instance that is the parent of the target menu. When the target menu does not belong to a MenuBar instance, this value is `undefined`.
- `menu` A reference to the Menu instance that is shown.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `form` is defined and passed to `myMenu.addEventListener()` as the second parameter. The event object is captured by the `menuShow` handler in the `evt` parameter. When the `menuShow` event is broadcast, a `trace` statement is sent to the Output panel.

```
form = new Object();
form.menuShow = function(evt){
    trace("Menu opened: "+evt.menu);
```

```
}
myMenu.addEventListener("menuShow", form);
```

See also

Menu.menuHide

Menu.removeAll()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenu.removeAll()

Parameters

None.

Returns

Nothing.

Description

Method; removes all items and refreshes the menu.

Example

The following example removes all nodes from the menu:

```
myMenu.removeAll();
```

Menu.removeMenuItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenuItem.removeMenuItem()

Returns

A reference to the returned menu item (XML node). This value is `undefined` if there is no item in that position.

Description

Method; removes the specified menu item and all its children, and refreshes the menu.

Example

The following example removes the menu item referenced by the variable `theItem`:

```
theItem.removeMenuItem();
```

Menu.removeMenuItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenu.removeMenuItemAt(index)
```

Parameters

index The index of the menu item to remove.

Returns

A reference to the returned menu item (XML node). This value is `undefined` if there is no item in that position.

Description

Method; removes the menu item and all its children at the specified index. If there is no menu item at that index, calling this method has no effect.

Example

The following example removes a menu item at index 3:

```
var item = myMenu.removeMenuItemAt(3);
```

Menu.rollOut

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.rollOut = function(eventObject){
  // insert your code here
}
myMenu.addEventListener("rollOut", listenerObject)
```

Description

Event; broadcast to all registered listeners when the pointer rolls off a menu item.

Version 2 components use a dispatcher-listener event model. When a Menu component broadcasts a `rollOut` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Menu.rollOut` event's event object has one additional property: `menuItem`, which is a reference to the menu item (XML node) that the pointer rolled off.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `form` is defined and passed to `myMenu.addEventListener()` as the second parameter. The event object is captured by the `rollOut` handler in the `evt` parameter. When the `rollOut` event is broadcast, a `trace` statement is sent to the Output panel.

```
form = new Object();
form.rollOut = function(evt){
    trace("Menu rollOut: "+evt.menuItem.attributes.label);
}
myMenu.addEventListener("rollOut", form);
```

Menu.rollOver

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.rollOver = function(eventObject){
  // insert your code here
}
myMenu.addEventListener("rollOver", listenerObject)
```

Description

Event; broadcast to all registered listeners when the pointer rolls over a menu item.

Version 2 components use a dispatcher-listener event model. When a Menu component broadcasts a `rollover` event, the event is handled by a function (also called a *handler*) that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Menu.rollOver` event's event object has one additional property: `menuItem`, which is a reference to the menu item (XML node) that the pointer rolled over.

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `form` is defined and passed to `myMenu.addEventListener()` as the second parameter. The event object is captured by the `rollOver` handler in the `evt` parameter. When the `rollOver` event is broadcast, a `trace` statement is sent to the Output panel.

```
form = new Object();
form.rollOver = function(evt){
    trace("Menu rollOver: "+evt.menuItem.attributes.label);
}
myMenu.addEventListener("rollOver", form);
```

Menu.setMenuItemEnabled()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenu`.setMenuItemEnabled(`*item*`, `*enable*`)`

Parameters

item An XML node; the target menu item's node in the data provider.

enable A Boolean value indicating whether the item is enabled (`true`) or not (`false`).

Returns

Nothing.

Description

Method; changes the target item's `enabled` attribute to the state specified in the *enable* parameter. If this call results in a change of state, the item is redrawn with the new state.

Example

The following example disables the second child of `myMenu`:

```
var myItem = myMenu.getMenuItemAt(1);
myMenu.setMenuItemEnabled(myItem, false);
```

See also

Menu.setMenuItemSelected()

Menu.setMenuItemSelected()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenu.setMenuItemSelected(*item*, *select*)

Parameters

item An XML node. The target menu item's node in the data provider.

select A Boolean value indicating whether the item is selected (`true`) or not (`false`). If the item is a check box, its check mark is visible or not visible. If a selected item is a radio button, it becomes the current selection in the radio group.

Returns

Nothing.

Description

Method; changes the `selected` attribute of the item to the state specified by the *select* parameter. If this call results in a change of state, the item is redrawn with the new state. This is only meaningful for items whose `type` attribute is set to `"radio"` or `"check"`, because it causes their dot or check to appear or disappear. If you call this method on an item whose type is `"normal"` or `"separator"`, it has no effect.

Example

The following example deselects the second child of `myMenu`:

```
var myItem = myMenu.getMenuItemAt(1);
myMenu.setMenuItemSelected(myItem, false);
```

Menu.show()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenu.show(*x*, *y*)

Parameters

x The *x* coordinate.

y The *y* coordinate.

Returns

Nothing.

Description

Method; opens a menu at a specific location. The menu is automatically resized so that all of its top-level items are visible, and the upper left corner is placed at the specified location in the coordinate system provided by the component's parent.

If the *x* and *y* parameters are omitted, the menu is shown at its previous location.

Example

The following example displays a menu 10 pixels down and to the right of the (0,0) origin point of the component's parent:

```
myMenu.show(10, 10);
```

See also

`Menu.hide()`

MenuDataProvider class

ActionScript Class Name mx.controls.menuclasses.MenuDataProvider

The MenuDataProvider class is a decorator (mix-in) class that adds functionality to the XMLNode global class. This functionality lets XML instances assigned to a `Menu.dataProvider` property use the MenuDataProvider methods and properties to manipulate their own data as well as the associated menu views.

Keep in mind these concepts about the MenuDataProvider class:

- MenuDataProvider is a decorator (mix-in) class. You do not need to instantiate it to use it.
- Menus natively accept XML as a `dataProvider` property value.
- If a Menu class is instantiated, all XML instances in the SWF file are decorated by the MenuDataProvider class.
- Only MenuDataProvider methods broadcast events to the Menu components. You can still use native XML methods, but they do not broadcast events that refresh the Menu views. To control the data model, use MenuDataProvider methods. For read-only operations like moving through the Menu hierarchy, use XML methods.
- All items in the Menu component are XML objects decorated with the MenuDataProvider class.
- Changes to item attributes are not reflected in the onscreen menu until redrawing occurs.

Method summary for the MenuDataProvider class

The following table lists the methods of the MenuDataProvider class.

Method	Description
`MenuDataProvider.addMenuItem()`	Adds a child item.
`MenuDataProvider.addMenuItemAt()`	Adds a child item at a specified location.
`MenuDataProvider.getMenuItemAt()`	Gets a reference to a menu item at a specified location.
`MenuDataProvider.indexOf()`	Returns the index of a specified menu item.
`MenuDataProvider.removeMenuItem()`	Removes a menu item.
`MenuDataProvider.removeMenuItemAt()`	Removes a menu item at a specified location.

MenuDataProvider.addMenuItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenuDataProvider.addMenuItem(initObject)
```

Usage 2:

```
myMenuDataProvider.addMenuItem(childMenuItem)
```

Parameters

`initObject` An object containing the attributes that initialize a Menu item's attributes. For more information, see "About menu item XML attributes" on page 541.

`childMenuItem` An XML node.

Returns

A reference to an XMLNode object.

Description

Method; Usage 1 adds a child item to the end of a parent menu item (which could be the menu itself). The menu item is constructed from the values passed in the *`initObject`* parameter. Usage 2 adds a child item that is defined in the specified XML *`childMenuItem`* parameter to the end of a parent menu item.

Any node or menu item in a MenuDataProvider instance can call the methods of the MenuDataProvider class.

Example

The following example adds a new node to a specified node in the menu:

```
var item0 = myMenuDP.getMenuItemAt(0);
item0.addMenuItem("Inbox", { label:"Item 1", icon:"radioItemIcon",
  type:"radio", selected:false, enabled:true, instanceName:"radioItem1",
  groupName:"myRadioGroup" } );
```

MenuDataProvider.addMenuItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenuDataProvider.addMenuItemAt(index, initObject)
```

Usage 2:

```
myMenuDataProvider.addMenuItemAt(index, childMenuItem)
```

Parameters

index An integer.

initObject An object containing the specific attributes that initialize a Menu item's attributes. For more information, see "About menu item XML attributes" on page 541.

childMenuItem An XML node.

Returns

A reference to the added XML node.

Description

Method; Usage 1 adds a child item at the specified index position in the parent menu item (which could be the menu itself). The menu item is constructed from the values passed in the *initObject* parameter. Usage 2 adds a child item that is defined in the specified XML *childMenuItem* parameter to the specified index of a parent menu item.

Any node or menu item in a MenuDataProvider instance can call the methods of the MenuDataProvider class.

Example

Usage 1: The following example adds a new node as the second child of the root of the menu:

```
myMenuDataProvider.addMenuItemAt(1, {label:"Item 1", type:"radio",
  selected:false, enabled:true, instanceName:"radioItem1",
  groupName:"myRadioGroup"});
```

MenuDataProvider.getMenuItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuDataProvider.getMenuItemAt(index)
```

Parameters

index An integer indicating the position of the menu.

Returns

A reference to the specified XML node.

Description

Method; returns a reference to the specified child menu item of the current menu item.

Any node or menu item in a MenuDataProvider instance can call the methods of the MenuDataProvider class.

Example

The following example finds the node you want to get, and then gets the second child of `myMenuItem`:

```
var myMenuItem = myMenuDP.firstChild.firstChild;
myMenuItem.getMenuItemAt(1);
```

MenuDataProvider.indexOf()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuDataProvider.indexOf(item)
```

Parameters

item A reference to the XML node that describes the menu item.

Returns

The index of the specified menu item; returns `undefined` if the item does not belong to this menu.

Description

Method; returns the index of the specified menu item in this parent menu item.

Any node or menu item in a MenuDataProvider instance can call the methods of the MenuDataProvider class.

Example

The following example adds a menu item to a parent item and gets the item's index:

```
var myMenuItem = myParentMenuItem.addMenuItem({label:"That item"});
var myIndex = myParentMenuItem.indexOf(myItem);
```

MenuDataProvider.removeMenuItem()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenuDataProvider.removeMenuItem()

Parameters

None.

Returns

A reference to the removed Menu item (XML node); `undefined` if an error occurs.

Description

Method; removes the target item and any child nodes.

Any node or menu item in a MenuDataProvider instance can call the methods of the MenuDataProvider class.

Example

The following example removes `myMenuItem` from its parent:

```
myMenuItem.removeMenuItem();
```

MenuDataProvider.removeMenuItemAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myMenuDataProvider.removeMenuItemAt(*index*)

Parameters

index The index of the menu item.

Returns

A reference to the removed menu item. This value is `undefined` if there is no item in that position.

Description

Method; removes the child item of the menu item specified by the *index* parameter. If there is no menu item at that index, calling this method has no effect.

Any node or menu item in a MenuDataProvider instance can call the methods of the MenuDataProvider class.

Example

The following example removes the fourth item:

```
myMenuDataProvider.removeMenuItemAt(3);
```

MenuBar component (Flash Professional only)

The MenuBar component lets you create a horizontal menu bar with pop-up menus and commands, just like the menu bars that contain File and Edit menus in common software applications. The MenuBar component complements the Menu component by providing a clickable interface to show and hide menus that behave as a group for mouse and keyboard interactivity.

The MenuBar component lets you create an application menu in a few steps. To build a menu bar, you can either assign an XML data provider to the menu bar that describes a series of menus, or use the `MenuBar.addMenu()` method to add menu instances one at a time.

Each menu in the menu bar is composed of two parts: the menu and the button that causes the menu to open (called the menu activator). These clickable menu activators appear in the menu bar as a text label with inset and outset border highlight states that react to interaction from the mouse and keyboard.

When a menu activator is clicked, the corresponding menu opens below it. The menu stays active until the activator is clicked again, or until a menu item is selected or a click occurs outside the menu area.

In addition to creating menu activators that show and hide menus, the MenuBar component creates group behavior among a series of menus. This lets a user scan a large number of command choices by rolling over the series of activators or by using the arrow keys to move through the lists. Mouse and keyboard interactivity work together to let the user jump from menu to menu in the menu bar.

A user cannot scroll through menus on a menu bar. If menus exceed the width of the menu bar, they are masked.

You cannot make the MenuBar component accessible to screen readers.

Interacting with the MenuBar component (Flash Professional only)

You can use the mouse and keyboard to interact with a MenuBar component.

Rolling over a menu activator displays an outset border highlight around the activator label.

When a MenuBar instance has focus either from clicking or tabbing, you can use the following keys to control it:

Key	Description
Down Arrow	Moves the selection down a menu row.
Up Arrow	Moves the selection up a menu row.
Right Arrow	Moves the selection to the next button.
Left Arrow	Moves the selection to the previous button.
Enter/Escape	Closes an open menu.

Note: If a menu is open, you can't press the Tab key to close it. You must either make a selection or close the menu by pressing Escape.

Using the MenuBar component (Flash Professional only)

You can use the MenuBar component to add a set of menus (for example, File, Edit, Special, Window) to the top edge of an application.

MenuBar parameters

You can set the following authoring parameter for each MenuBar component instance in the Property inspector or in the Component inspector:

Labels An array that adds menu activators with the specified labels to the MenuBar component. The default value is `[]` (an empty array).

You cannot access the Labels parameter using ActionScript. However, you can write ActionScript to control additional options for the MenuBar component using its properties, methods, and events. For more information, see "MenuBar class (Flash Professional only)" on page 578.

Creating an application with the MenuBar component

In this example, you drag a MenuBar component to the Stage, add code to fill the instance with menus, and attach listeners to the menus to respond to menu item selection.

To use a MenuBar component in an application:

1. Select File > New and create a new Flash document.
2. Drag the MenuBar component from the Components panel to the Stage.
3. Position the menu at the top of the Stage for a standard layout.
4. Select the MenuBar instance, and in the Property inspector, enter the instance name **myMenuBar**.
5. In the Actions panel on Frame 1, enter the following code:

```
var menu = myMenuBar.addMenu("File");
menu.addMenuItem({label:"New", instanceName:"newInstance"});
menu.addMenuItem({label:"Open", instanceName:"openInstance"});
menu.addMenuItem({label:"Close", instanceName:"closeInstance"});
```

   This code adds a File menu to the MenuBar instance. It then uses a Menu method to add three menu items: New, Open, and Close.

6. In the Actions panel on Frame 1, enter the following code:

```
var listen = new Object();
listen.change = function(evt){
  var menu = evt.menu;
  var item = evt.menuItem
  if (item == menu.newInstance){
    myNew();
    trace(item);
  }else if (item == menu.openInstance){
    myOpen()
    trace(item);
  }
}
menu.addEventListener("change",listen);
```

This code creates a listener object, `listen`, that uses the event object, `evt`, to catch menu item selections.

Note: You must call the `addEventListener()` method to register the listener with the menu instance, not with the menu bar instance.

7. Select Control > Test Movie to test the MenuBar component.

Customizing the MenuBar component (Flash Professional only)

This component sizes itself according to the activator labels that are supplied through the `dataProvider` property or the methods of the MenuBar class. When an activator button is in a menu bar, it remains at a fixed size that is dependent on the font styles and the text length.

Using styles with the MenuBar component

The MenuBar component creates an activator label for each menu in a group. You can use styles to change the look of the activator labels. A MenuBar component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.

The MenuBar component also forwards all style settings for Menu style properties to the composed Menu instances. For a list of Menu style properties, see "Using styles with the Menu component" on page 548.

Using skins with the MenuBar component

The MenuBar component uses three skins to represent its background, uses a movie clip symbol for highlighting individual items, and contains a Menu component as the pop-up which itself is skinnable. The MenuBar skins are described below. For information on skinning the Menu component, see "Using skins with the Menu component" on page 550.

The MenuBar component supports the following skin properties.

Property	Description
`menuBarBackLeftName`	The up state of the pop-up icon.
`menuBarBackRightName`	The down state of the pop-up icon.
`menuBarBackMiddleName`	The disabled state of the pop-up icon.

To create movie clip symbols for MenuBar skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the MenuBar Assets folder to the library for your document.
4. Expand the MenuBar Assets/Elements folder in the library of your document.
5. Open the symbols you want to customize for editing.

   For example, open the MenuBarBackLeft symbol.
6. Customize the symbol as desired.

   For example, change the outer edge to blank.
7. Repeat steps 5-6 for all symbols you want to customize.

   For example, set the outer edges for the middle and right symbols to black.
8. Click the Back button to return to the main Timeline.
9. Drag a MenuBar component to the Stage.
10. Set MenuBar properties so that they display items on the bar.
11. Select Control > Test Movie.

Note: The border used to highlight individual items in a MenuBar component is an instance of ActivatorSkin found in the Flash UI Components 2/Themes/MMDefault/Button Assets folder. This symbol can be customized to point to a different class to provide a different border. However, the symbol name cannot be modified, and you cannot use a different symbol for different MenuBar instances in a single document.

MenuBar class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > MenuBar

ActionScript Class Name mx.controls.MenuBar

The methods and properties of the MenuBar class let you create a horizontal menu bar with pop-up menus and commands. These methods and properties complement those of the Menu class by allowing you to create a clickable interface to show and hide menus that behave as a group for mouse and keyboard interactivity.

Method summary for the MenuBar class

The following table lists methods of the MenuBar class.

Method	Description
`MenuBar.addMenu()`	Adds a menu to the menu bar.
`MenuBar.addMenuAt()`	Adds a menu at a specified location to the menu bar.
`MenuBar.getMenuAt()`	Gets a reference to a menu at a specified location.
`MenuBar.getMenuEnabledAt()`	Returns a Boolean value indicating whether a menu is enabled (`true`) or not (`false`).
`MenuBar.removeMenuAt()`	Removes a menu at a specified location from a menu bar.
`MenuBar.setMenuEnabledAt()`	A Boolean value indicating whether a menu is can be chosen (`true`) or not (`false`).

Methods inherited from the UIObject class

The following table lists the methods the MenuBar class inherits from the UIObject class. When calling these methods from the MenuBar object, use the form `MenuBar.`*`methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the MenuBar class inherits from the UIComponent class. When calling these methods from the MenuBar object, use the form `MenuBar.methodName`.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Property summary for the MenuBar class

The following table lists properties of the MenuBar class.

Property	Description
`MenuBar.dataProvider`	The data model for a menu bar.
`MenuBar.labelField`	A string that determines which attribute of each XMLNode to use as the label text of the menu.
`MenuBar.labelFunction`	A function that determines what to display in each menu's label.

Properties inherited from the UIObject class

The following table lists the properties the MenuBar class inherits from the UIObject class. When calling these properties from the MenuBar object, use the form `MenuBar.propertyName`.

Property	Description
`UIObject.bottom`	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
`UIObject.height`	The height of the object, in pixels. Read-only.
`UIObject.left`	The left edge of the object, in pixels. Read-only.
`UIObject.right`	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
`UIObject.scaleX`	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
`UIObject.scaleY`	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
`UIObject.top`	The position of the top edge of the object, relative to its parent. Read-only.
`UIObject.visible`	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
`UIObject.width`	The width of the object, in pixels. Read-only.
`UIObject.x`	The left edge of the object, in pixels. Read-only.
`UIObject.y`	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the MenuBar class inherits from the UIComponent class. When calling these properties from the MenuBar object, use the form `MenuBar.propertyName`.

Property	Description
`UIComponent.enabled`	Indicates whether the component can receive focus and input.
`UIComponent.tabIndex`	A number indicating the tab order for a component in a document.

Event summary for the MenuBar class

There are no events exclusive to the MenuBar class.

Events inherited from the Menu class

The following table lists the events the MenuBar class inherits from the Menu class. When calling these events from the MenuBar object, use the form `MenuBar.eventName`.

Event	Description
`Menu.change`	Broadcast when a user causes a change in a menu.
`Menu.menuHide`	Broadcast when a menu closes.
`Menu.menuShow`	Broadcast when a menu opens.
`Menu.rollOut`	Broadcast when the pointer rolls off an item.
`Menu.rollOver`	Broadcast when the pointer rolls over an item.

Events inherited from the UIObject class

The following table lists the events the MenuBar class inherits from the UIObject class. When calling these events from the MenuBar object, use the form `MenuBar.eventName`.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the MenuBar class inherits from the UIComponent class. When calling these events from the MenuBar object, use the form `MenuBar.eventName`.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

MenuBar.addMenu()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenuBar.addMenu(label)
```

Usage 2:

```
myMenuBar.addMenu(label, menuDataProvider)
```

Parameters

label A string indicating the label of the new menu.

menuDataProvider An XML or XMLNode instance that describes the menu and its items. If the value is an XML instance, the instance's first child is used.

Returns

A reference to the new Menu object.

Description

Method; Usage 1 adds a single menu and menu activator at the end of the menu bar and uses the specified label. Usage 2 adds a single menu and menu activator that are defined in the specified XML *menuDataProvider* parameter.

Example

Usage 1: The following example adds a File menu and then uses `Menu.addMenuItem()` to add the menu items New and Open:

```
var myMenuBar:mx.controls.MenuBar;
var myMenu:mx.controls.Menu;
```

```
myMenu = myMenuBar.addMenu("File");
myMenu.addMenuItem({label:"New", instanceName:"newInstance"});
myMenu.addMenuItem({label:"Open", instanceName:"openInstance"})
```

Usage 2: The following example adds a Font menu with the menu items Bold and Italic that are defined in the MenuDataProvider instance `myMenuDP2`:

```
var myMenuDP2 = new XML();
myMenuDP2.addMenuItem({type:"check", label:"Bold", instanceName:"check1"});
myMenuDP2.addMenuItem({type:"check", label:"Italic", instanceName:"check2"});
menu = myMenuBar.addMenu("Font",myMenuDP2);
```

MenuBar.addMenuAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenuBar.addMenuAt(index, label)
```

Usage 2:

```
myMenuBar.addMenuAt(index, label, menuDataProvider)
```

Parameters

`index` An integer indicating the position where the menu should be inserted. The first position is 0. To append to the end of the menu, call `MenuBar.addMenu(label)`.

`label` A string indicating the label of the new menu.

`menuDataProvider` An XML or XMLNode instance that describes the menu. If the value is an XML instance, the instance's first child is used.

Returns

A reference to the new Menu object.

Description

Method; Usage 1 adds a single menu and menu activator at the specified index with the specified label. Usage 2 adds a single menu and a labeled menu activator at the specified index. The content for the menu is defined in the *`menuDataProvider`* parameter.

Example

Usage 1: The following example places a menu to the left of all MenuBar menus:

```
menu = myMenuBar.addMenuAt(0,"Toreador");
menu.addMenuItem("About Macromedia Flash", instanceName:"aboutInst");
menu.addMenuItem("Preferences", instanceName:"PrefInst");
```

Usage 2: The following example adds an Edit menu with the menu items Undo, Redo, Cut, and Copy, which are defined in the MenuDataProvider instance `myMenuDP`:

```
var myMenuDP = new XML();
myMenuDP.addMenuItem({label:"Undo", instanceName:"undoInst"});
myMenuDP.addMenuItem({label:"Redo", instanceName:"redoInst"});
myMenuDP.addMenuItem({type:"separator"});
myMenuDP.addMenuItem({label:"Cut", instanceName:"cutInst"});
myMenuDP.addMenuItem({label:"Copy", instanceName:"copyInst"});

myMenuBar.addMenuAt(0,"Edit",myMenuDP);
```

MenuBar.dataProvider

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.dataProvider
```

Description

Property; the data model for items in a MenuBar component.

`MenuBar.dataProvider` is an XML node object. Setting this property replaces the existing data model of the MenuBar component. Whatever child nodes the data provider might have are used as the items for the menu bar itself; any subnodes of these child nodes are used as the items for their respective menus.

The default value is `undefined`.

Note: All XML or XMLNode instances are automatically given the methods and properties of the MenuDataProvider class when they are used with the MenuBar component.

Example

The following example imports an XML file and assigns it to the `MenuBar.dataProvider` property:

```
var myMenuBarDP = new XML();
myMenuBarDP.load("http://myServer.myDomain.com/source.xml");
myMenuBarDP.onLoad = function(success){
  if(success){
    myMenuBar.dataProvider = myMenuBarDP;
  } else {
  trace("error loading XML file");
  }
}
```

MenuBar.getMenuAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.getMenuAt(index)
```

Parameters

index An integer indicating the position of the menu.

Returns

A reference to the menu at the specified index. This value is `undefined` if there is no menu at that position.

Description

Method; returns a reference to the menu at the specified index.

Example

Because `getMenuAt()` returns an instance, it is possible to add items to a menu at the specified index. In the following example, after using the Labels authoring parameter to create the menu activators File, Edit, and View, the following code adds New and Open items to the File menu at runtime:

```
menu = myMenuBar.getMenuAt(0);
menu.addMenuItem({label:"New",instanceName:"newInst"});
menu.addMenuItem({label:"Open",instanceName:"openInst"});
```

MenuBar.getMenuEnabledAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.getMenuEnabledAt(index)
```

Parameters

index The index of the menu in the menu bar.

Returns

A Boolean value that indicates whether this menu can be chosen (`true`) or not (`false`).

Description

Method; returns a Boolean value that indicates whether this menu can be chosen (`true`) or not (`false`).

Example

The following example calls the method on the menu in the first position of `myMenuBar`:

```
myMenuBar.getMenuEnabledAt(0);
```

MenuBar.labelField

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.labelField
```

Description

Property; a string that determines which attribute of each XML node to use as the label text of the menu. This property is also passed to any menus that are created from the menu bar. The default value is `"label"`.

After the `dataProvider` property is set, this property is read-only.

Example

The following example uses the `name` attribute of each node as the label text:

```
myMenuBar.labelField = "name";
```

MenuBar.labelFunction

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.labelFunction
```

Description

Property; a function that determines what to display in each menu's label text. The function accepts the XML node associated with an item as a parameter and returns a string to be used as label text. This property is passed to any menus created from the menu bar. The default value is `undefined`.

After the `dataProvider` property is set, this property is read-only.

Example

The following example of a label function builds and returns a custom label from the node attributes:

```
myMenuBar.labelFunction = function(node){
var a = node.attributes;
return "The Price for " + a.name + " is " + a.price;
};
```

MenuBar.removeMenuAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.removeMenuAt(index)
```

Parameters

index The index of the menu to be removed from the menu bar.

Returns

A reference to the menu at the specified index in the menu bar. This value is `undefined` if there is no menu in that position in the menu bar.

Description

Method; removes the menu at the specified index. If there is no menu item at that index, calling this method has no effect.

Example

The following example removes the menu at index 4:

```
myMenuBar.removeMenuAt(4);
```

MenuBar.setMenuEnabledAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.setMenuEnabledAt(index, boolean)
```

Parameters

index The index of the menu item to set in the MenuBar instance.

boolean A Boolean value indicating whether the menu item at the specified index is enabled (`true`) or not (`false`).

Returns

Nothing.

Description

Method; enables the menu at the specified index. If there is no menu at that index, calling this method has no effect.

Example

The following example enables the item at index 3 in the MenuBar object `myMenuBar`:

```
myMenuBar.setMenuEnabledAt(3, true);
```

NumericStepper component

The NumericStepper component allows a user to step through an ordered set of numbers. The component consists of a number in a text box displayed beside small up and down arrow buttons. When a user presses the buttons, the number is raised or lowered incrementally according to the unit specified in the `stepSize` parameter, until the user releases the buttons or until the maximum or minimum value is reached. The text in the NumericStepper component's text box is also editable.

The NumericStepper component handles only numeric data. Also, you must resize the stepper while authoring to display more than two numeric places (for example, the numbers 5246 or 1.34).

A stepper can be enabled or disabled in an application. In the disabled state, a stepper doesn't receive mouse or keyboard input. An enabled stepper receives focus if you click it or tab to it and its internal focus is set to the text box. When a NumericStepper instance has focus, you can use the following keys control it:

Key	Description
Down Arrow	Value changes by one unit.
Left Arrow	Moves the insertion point to the left within the text box.
Right Arrow	Moves the insertion point to the right within the text box.
Shift+Tab	Moves focus to the previous object.
Tab	Moves focus to the next object.
Up Arrow	Value changes by one unit.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each stepper instance reflects the setting of the value parameter in the Property inspector or Component inspector during authoring. However, there is no mouse or keyboard interaction with the stepper's arrow buttons in the live preview.

When you add the NumericStepper component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code to enable accessibility:

```
mx.accessibility.NumericStepperAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the NumericStepper component

You can use the NumericStepper anywhere you want a user to select a numeric value. For example, you could use a NumericStepper component in a form to allow a user to set a credit card expiration date. You could also use a NumericStepper component to allow a user to increase or decrease a font size.

NumericStepper parameters

You can set the following authoring parameters for each NumericStepper instance in the Property inspector or in the Component inspector:

value sets the value displayed in the text area of the stepper. The default value is 0.

minimum sets the minimum value that can be displayed in the stepper. The default value is 0.

maximum sets the maximum value that can be displayed in the stepper. The default value is 10.

stepSize sets the unit by which the stepper increases or decreases with each click. The default value is 1.

You can write ActionScript to control these and additional options for the NumericStepper component using its properties, methods, and events. For more information, see "NumericStepper class" on page 592.

Creating an application with the NumericStepper component

The following procedure explains how to add a NumericStepper component to an application while authoring. In this example, the stepper allows a user to pick a movie rating from 0 to 5 stars with half-star increments.

To create an application with the NumericStepper component:

1. Drag a NumericStepper component from the Components panel to the Stage.
2. In the Property inspector, enter the instance name **starStepper**.
3. In the Property inspector, do the following:
   - Enter 0 for the minimum parameter.
   - Enter 5 for the maximum parameter.
   - Enter .5 for the stepSize parameter.
   - Enter 0 for the value parameter.
4. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
movieRate = new Object();
movieRate.change = function (eventObject){
  starChart.value = eventObject.target.value;
}
starStepper.addEventListener("change", movieRate);
```

   The last line of code adds a `change` event handler to the `starStepper` instance. The handler sets the `starChart` movie clip to display the amount of stars indicated by the `starStepper` instance. (To see this code work, you must create a `starChart` movie clip with a `value` property that displays the stars.)

Customizing the NumericStepper component

You can transform a NumericStepper component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`) or any applicable properties and methods of the NumericStepper class. (See "NumericStepper class" on page 592.)

Resizing the NumericStepper component does not change the size of the down and up arrow buttons. If the stepper is resized to be greater than the default height, the arrow buttons are pinned to the top and bottom of the component. The arrow buttons always appear to the right of the text box.

Using styles with the NumericStepper component

You can set style properties to change the appearance of a NumericStepper instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

A NumericStepper component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textAlign`	Both	The text alignment: either `"left"`, `"right"`, or `"center"`. The default value is `"center"`.

Style	Theme	Description
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".
repeatDelay	Both	The number of milliseconds of delay between when a user first presses a button and when the action begins to repeat. The default value is 500 (half a second).
repeatInterval	Both	The number of milliseconds between automatic clicks when a user holds the mouse button down on a button. The default value is 35.
symbolColor	Sample	The color of the arrows. The default value is 0x2B333C (dark gray).

Using skins with the NumericStepper component

The NumericStepper component uses skins to represent its up and down button states. To skin the NumericStepper component while authoring, modify skin symbols in the Flash UI Components 2/Themes/MMDefault/Stepper Assets/States folder in the library. For more information, see "About skinning components" on page 80.

If a stepper is enabled, the down and up buttons display their over states when the pointer moves over them. The buttons display their down state when pressed. The buttons return to their over state when the mouse is released. If the pointer moves off the buttons while the mouse is pressed, the buttons return to their original state.

If a stepper is disabled, it displays its disabled state, regardless of user interaction.

A NumericStepper component supports the following skin properties:

Property	Description
upArrowUp	The up arrow button's up state. The default value is StepUpArrowUp.
upArrowDown	The up arrow button's pressed state. The default value is StepUpArrowDown.
upArrowOver	The up arrow button's over state. The default value is StepUpArrowOver.
upArrowDisabled	The up arrow button's disabled state. The default value is StepUpArrowDisabled.
downArrowUp	The down arrow button's up state. The default value is StepDownArrowUp.
downArrowDown	The down arrow button's down state. The default value is StepDownArrowDown.
downArrowOver	The down arrow button's over state. The default value is StepDownArrowOver.
downArrowDisabled	The down arrow button's disabled state. The default value is StepDownArrowDisabled.

To create movie clip symbols for NumericStepper skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the Stepper Assets folder to the library for your document.
4. Expand the Stepper Assets folder in the library of your document.
5. Expand the Stepper Assets/States folder in the library of your document.
6. Open the symbols you want to customize for editing.

   For example, open the StepDownArrowDisabled symbol.
7. Customize the symbol as desired.

   For example, change the white inner graphics to a light gray.
8. Repeat steps 6-7 for all symbols you want to customize.

   For example, repeat the same change on the up arrow.
9. Click the Back button to return to the main Timeline.
10. Drag a NumericStepper component to the Stage.

    This example has customized the disabled skins, so use ActionScript to set the NumericStepper instance to be disabled in order to see the modified skins.
11. Select Control > Test Movie.

Note: The Stepper Assets/States folder also contains a StepTrack symbol, which is used as a spacer between the up and down skins if the total height of the NumericStepper instance is greater than the sum of the two arrow heights. This symbol linkage identifier is not available for modification through a skin property, but the library symbol can be modified provided the linkage identifier remains unchanged.

NumericStepper class

Inheritance MovieClip > UIObject class > UIComponent class > NumericStepper

ActionScript Class Name mx.controls.NumericStepper

The properties of the NumericStepper class let you set the following at runtime: the minimum and maximum values displayed in the stepper, the unit by which the stepper increases or decreases in response to a click, and the current value displayed in the stepper.

Setting a property of the NumericStepper class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

The NumericStepper component uses the Focus Manager to override the default Flash Player focus rectangle and draw a custom focus rectangle with rounded corners. For more information, see "Creating custom focus navigation" on page 50.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.NumericStepper.version);
```

Note: The code `trace(myNumericStepperInstance.version);` returns `undefined`.

Method summary for the NumericStepper class

There are no methods exclusive to the NumericStepper class.

Methods inherited from the UIObject class

The following table lists the methods the NumericStepper class inherits from the UIObject class. When calling these methods from the NumericStepper object, use the form `NumericStepper.methodName`.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the NumericStepper class inherits from the UIComponent class. When calling these methods from the NumericStepper object, use the form `NumericStepper.methodName`.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Property summary for the NumericStepper class

The following table lists properties of the NumericStepper class.

Property	Description
NumericStepper.maximum	A number indicating the maximum range value.
NumericStepper.minimum	A number indicating the minimum range value.
NumericStepper.nextValue	A number indicating the next sequential value. This property is read-only.
NumericStepper.previousValue	A number indicating the previous sequential value. This property is read-only.
NumericStepper.stepSize	A number indicating the unit of change for each click.
NumericStepper.value	A number indicating the current value of the stepper.

Properties inherited from the UIObject class

The following table lists the properties the NumericStepper class inherits from the UIObject class. When calling these properties from the NumericStepper object, use the form `NumericStepper.`*`propertyName`*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the NumericStepper class inherits from the UIComponent class. When calling these properties from the NumericStepper object, use the form `NumericStepper.propertyName`.

Property	Description
`UIComponent.enabled`	Indicates whether the component can receive focus and input.
`UIComponent.tabIndex`	A number indicating the tab order for a component in a document.

Event summary for the NumericStepper class

The following table lists the event of the NumericStepper class.

Event	Description
`NumericStepper.change`	Triggered when the value of the stepper changes.

Events inherited from the UIObject class

The following table lists the events the NumericStepper class inherits from the UIObject class. When calling these events from the NumericStepper object, use the form `NumericStepper.eventName`.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the NumericStepper class inherits from the UIComponent class. When calling these events from the NumericStepper object, use the form `NumericStepper.eventName`.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

NumericStepper.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(click){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  ...
}
stepperInstance.addEventListener("change", listenerObject)
```

Description

Event; broadcast to all registered listeners when the value of the stepper is changed.

The first usage example uses an `on()` handler and must be attached directly to a NumericStepper instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the stepper `myStepper`, sends "_level0.myStepper" to the Output panel:

```
on(click){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`stepperInstance`*) dispatches an event (in this case, `change`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a stepper called `myNumericStepper` is changed. The first line of code creates a listener object called `form`. The second line defines a function for the `change` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function, in this example `eventObj`, to generate a message. The `target` property of an event object is the component that generated the event—in this example, `myNumericStepper`. The `NumericStepper.value` property is accessed from the event object's `target` property. The last line calls `EventDispatcher.addEventListener()` from `myNumericStepper` and passes it the `change` event and the `form` listener object as parameters.

```
form = new Object();
form.change = function(eventObj){
  // eventObj.target is the component that generated the change event,
  // i.e., the numeric stepper.
  trace("Value changed to " + eventObj.target.value);
}
myNumericStepper.addEventListener("change", form);
```

NumericStepper.maximum

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

stepperInstance.maximum

Description

Property; the maximum range value of the stepper. This property can contain a number of up to three decimal places. The default value is 10.

Example

The following example sets the maximum value of the stepper range to 20:

```
myStepper.maximum = 20;
```

See also

`NumericStepper.minimum`

NumericStepper.minimum

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

stepperInstance.minimum

Description

Property; the minimum range value of the stepper. This property can contain a number of up to three decimal places. The default value is 0.

Example

The following example sets the minimum value of the stepper range to 100:

```
myStepper.minimum = 100;
```

See also

NumericStepper.maximum

NumericStepper.nextValue

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

stepperInstance.nextValue

Description

Property (read-only); the next sequential value. This property can contain a number of up to three decimal places.

Example

The following example sets the `stepSize` property to 1 and the starting value to 4, which would make the value of `nextValue` 5:

```
myStepper.stepSize = 1;
myStepper.value = 4;
trace(myStepper.nextValue);
```

See also

NumericStepper.previousValue

NumericStepper.previousValue

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

stepperInstance.previousValue

Description

Property (read-only); the previous sequential value. This property can contain a number of up to three decimal places.

Example

The following example sets the `stepSize` property to 1 and the starting value to 4, which would make the value of `nextValue` 3:

```
myStepper.stepSize = 1;
myStepper.value = 4;
trace(myStepper.previousValue);
```

See also

NumericStepper.nextValue

NumericStepper.stepSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

stepperInstance.stepSize

Description

Property; the unit amount to change from the current value. The default value is 1. This value cannot be 0. This property can contain a number of up to three decimal places.

Example

The following example sets the current `value` property to 2 and the `stepSize` unit to 2. The value of `nextValue` is 4:

```
myStepper.value = 2;
myStepper.stepSize = 2;
trace(myStepper.nextValue);
```

NumericStepper.value

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
stepperInstance.value
```

Description

Property; the current value displayed in the text area of the stepper. The value is not assigned if it does not correspond to the stepper's range and step increment as defined in the `stepSize` property. This property can contain a number of up to three decimal places.

Example

The following example sets the current value of the stepper to 10 and sends the value to the Output panel:

```
myStepper.value = 10;
trace(myStepper.value);
```

PopUpManager class

ActionScript Class Name mx.managers.PopUpManager

The PopUpManager class lets you create overlapping windows that can be modal or nonmodal. (A modal window doesn't allow interaction with other windows while it's active.) You use the methods of this class to create and destroy pop-up windows.

Method summary for the PopUpManager class

The following table lists the methods of the PopUpManager class.

Method	Description
`PopUpManager.createPopUp()`	Creates a pop-up window.
`PopUpManager.deletePopUp()`	Deletes a pop-up window created by a call to `PopUpManager.createPopUp()`.

PopUpManager.createPopUp()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
PopUpManager.createPopUp(parent, class, modal [, initobj, outsideEvents])
```

Parameters

parent A reference to a window to pop-up over.

class A reference to the class of object you want to create.

modal A Boolean value indicating whether the window is modal (`true`) or not (`false`).

initobj An object containing initialization properties. This parameter is optional.

outsideEvents A Boolean value indicating whether an event is triggered if the user clicks outside the window (`true`) or not (`false`). This parameter is optional.

Returns

A reference to the window that was created.

Description

Method; if modal, a call to `createPopUp()` finds the topmost parent window starting with parent and creates an instance of class. If nonmodal, a call to `createPopUp()` creates an instance of the class as a child of the parent window.

Example

The following code creates a modal window when the button is clicked:

```
lo = new Object();
lo.click = function(){
  mx.managers.PopUpManager.createPopUp(_root, mx.containers.Window, true);
}
button.addEventListener("click", lo);
```

PopUpManager.deletePopUp()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004

Usage

```
windowInstance.deletePopUp();
```

Parameters

None.

Returns

Nothing.

Description

Method; deletes a pop-up window and removes the modal state. It is the responsibility of the overlapped window to call `PopUpManager.deletePopUp()` when the window is being destroyed.

Example

The following code creates a modal window named `win` with a close button, and deletes the window when the close button is clicked:

```
import mx.managers.PopUpManager
import mx.containers.Window
win = PopUpManager.createPopUp(_root, Window, true, {closeButton:true});
lo = new Object();
lo.click = function(){
  win.deletePopUp();
}
win.addEventListener("click", lo);
```

ProgressBar component

The ProgressBar component displays the progress of loading content. The loading process can be determinate or indeterminate. A *determinate* progress bar is a linear representation of a task's progress over time and is used when the amount of content to load is known. An *indeterminate* progress bar is used when the amount of content to load is unknown. You can add a label to display the progress of the loading content.

By default, components are set to export in the first frame. This means that components are loaded into an application before the first frame is rendered. If you want to create a preloader for an application, you must deselect Export in First Frame in each component's Linkage Properties dialog box (available from the Library options menu). The progress bar, however, should be set to Export in First Frame, because it must appear first, while other content streams into Flash Player.

The ProgressBar component contains a left cap, a right cap, and a progress track. The caps are simply the ends of the progress bar, where the progress track visually ends. A live preview of each ProgressBar instance reflects changes made to parameters in the Property inspector or Component inspector during authoring. The following parameters are reflected in the live preview: conversion, direction, label, labelPlacement, mode, and source.

Using the ProgressBar component

A progress bar lets you display the progress of content as it loads. This is essential feedback for users as they interact with an application.

There are several modes in which to use the ProgressBar component; you set the mode with the mode parameter. The most commonly used modes are event mode and polled mode. These modes use the source parameter to specify a loading process that either emits `progress` and `complete` events (event mode), or exposes `getBytesLoaded()` and `getsBytesTotal()` methods (polled mode). You can also use the ProgressBar component in manual mode by manually setting the `maximum`, `minimum`, and `indeterminate` properties along with calls to the `ProgressBar.setProgress()` method.

ProgressBar parameters

You can set the following authoring parameters for each ProgressBar instance in the Property inspector or in the Component inspector:

mode is the mode in which the progress bar operates. This value can be one of the following: `event`, `polled`, or `manual`. The default value is `event`.

source is a string to be converted into an object representing the instance name of the source.

direction indicates the direction toward which the progress bar fills. This value can be `right` or `left`; the default value is `right`.

label is the text indicating the loading progress. This parameter is a string in the format "%1 out of %2 loaded (%3%%)". In this string, %1 is a placeholder for the current bytes loaded, %2 is a placeholder for the total bytes loaded, and %3 is a placeholder for the percent of content loaded. The characters "%%" are a placeholder for the "%" character. If a value for %2 is unknown, it is replaced by two question marks (??). If a value is undefined, the label doesn't display.

labelPlacement indicates the position of the label in relation to the progress bar. This parameter can be one of the following values: `top`, `bottom`, `left`, `right`, `center`. The default value is `bottom`.

conversion is a number by which to divide the %1 and %2 values in the label string before they are displayed. The default value is 1.

You can write ActionScript to control these and additional options for the ProgressBar component using its properties, methods, and events. For more information, see "ProgressBar class" on page 607.

Creating an application with the ProgressBar component

The following procedure explains how to add a ProgressBar component to an application while authoring. In this example, the progress bar is used in event mode. In event mode, the loading content must emit `progress` and `complete` events that the progress bar uses to display progress. (These events are emitted by the Loader component. For more information, see "Loader component" on page 484.)

To create an application with the ProgressBar component in event mode:

1. Drag a ProgressBar component from the Components panel to the Stage.
2. In the Property inspector, do the following:
   - Enter the instance name **pBar**.
   - Select Event for the mode parameter.
3. Drag a Loader component from the Components panel to the Stage.
4. In the Property inspector, enter the instance name **loader**.
5. Select the progress bar on the Stage and, in the Property inspector, enter **loader** for the source parameter.
6. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code, which loads a JPEG file into the Loader component:

```
loader.autoLoad = false;
loader.contentPath = "http://imagecache2.allposters.com/images/86/
  017_PP0240.jpg";
pBar.source = loader;
// loading does not start until load() is invoked
loader.load();
```

In the following example, the progress bar is used in polled mode. In polled mode, the ProgressBar uses the `getBytesLoaded()` and `getBytesTotal()` methods of the source object to display its progress.

To create an application with the ProgressBar component in polled mode:

1. Drag a ProgressBar component from the Components panel to the Stage.
2. In the Property inspector, do the following:
   - Enter the instance name **pBar**.
   - Select Polled for the mode parameter.
   - Enter **loader** for the source parameter.
3. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code, which creates a Sound object called `loader` and calls `loadSound()` to load a sound into the Sound object:

```
var loader:Object = new Sound();
loader.loadSound("http://soundamerica.com/sounds/sound_fx/A-E/air.wav",
  true);
```

In the following example, the progress bar is used in manual mode. In manual mode, you must set the `maximum`, `minimum`, and `indeterminate` properties in conjunction with the `setProgress()` method to display progress. You do not set the `source` property in manual mode.

To create an application with the ProgressBar component in manual mode:

1. Drag a ProgressBar component from the Components panel to the Stage.
2. In the Property inspector, do the following:
   - Enter the instance name **pBar**.
   - Select Manual for the mode parameter.
3. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code, which updates the progress bar manually on every file download by using calls to `setProgress()`:

```
for(var:Number i=1; i <= total; i++){
  // insert code to load file
pBar.setProgress(i, total);
}
```

Customizing the ProgressBar component

You can transform a ProgressBar component horizontally while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use `UIObject.setSize()`.

The progress bar's left cap, right cap, and track graphic are set at a fixed size. When you resize a progress bar, its middle portion is resized to fit between the two caps. If a progress bar is too small, it may not render correctly.

Using styles with the ProgressBar component

You can set style properties to change the appearance of a progress bar instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

A ProgressBar component supports the following styles:

Style	Theme	Description
themeColor	Halo	The base color scheme of a component. Possible values are "haloGreen", "haloBlue", and "haloOrange". The default value is "haloGreen".
color	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
disabledColor	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
embedFonts	Both	A Boolean value that indicates whether the font specified in fontFamily is an embedded font. This style must be set to true if fontFamily refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to true and fontFamily does not refer to an embedded font, no text will be displayed. The default value is false.
fontFamily	Both	The font name for text. The default value is "_sans".
fontSize	Both	The point size for the font. The default value is 10.
fontStyle	Both	The font style: either "normal" or "italic". The default value is "normal".
fontWeight	Both	The font weight: either "none" or "bold". The default value is "none". All components can also accept the value "normal" in place of "none" during a setStyle() call, but subsequent calls to getStyle() will return "none".
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".
barColor	Sample	The foreground color in denoting the percent complete. The default color is white. To set the bar color on a Halo-themed component, set the themeColor style property.
trackColor	Sample	The background color for the bar. The default value is 0x666666 (dark gray).

Using skins with the ProgressBar component

The ProgressBar component uses skins to represent the progress bar track, the completed bar, and an indeterminate bar. To skin the ProgressBar component while authoring, modify symbols in the Flash UI Components 2/Themes/MMDefault/ProgressBar Elements folder. For more information, see "About skinning components" on page 80.

The track and bar graphics are each made up of three skins corresponding to the left and right caps and the middle. The caps are used "as is," and the middle is resized horizontally to fit the width of the ProgressBar instance.

The indeterminate bar is used when the ProgressBar instance's indeterminate property is set to true. The skin is resized horizontally to fit the width of the progress bar.

A ProgressBar component supports the following skin properties:

Property	Description
progTrackMiddleName	The expandable middle of the track. The default value is ProgTrackMiddle.
progTrackLeftName	The fixed-size left cap. The default value is ProgTrackLeft.
progTrackRightName	The fixed-size right cap. The default value is ProgTrackRight.
progBarMiddleName	The expandable middle bar graphic. The default value is ProgBarMiddle.
progBarLeftName	The fixed-size left bar cap. The default value is ProgBarLeft.
progBarRightName	The fixed-size right bar cap. The default value is ProgBarRight.
progIndBarName	The indeterminate bar graphic. The default value is ProgIndBar.

To create movie clip symbols for ProgressBar skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the ProgressBar Assets folder to the library for your document.
4. Expand the ProgressBar Assets/Elements folder in the library of your document.
5. Open the symbols you want to customize for editing.

   For example, open the ProgIndBar symbol.
6. Customize the symbol as desired.

   For example, flip the track horizontally.
7. Repeat steps 5-6 for all symbols you want to customize.
8. Click the Back button to return to the main Timeline.
9. Drag a ProgressBar component to the Stage.

   To view the skins modified in this example, use ActionScript to set the indeterminate property to true.
10. Select Control > Test Movie.

ProgressBar class

Inheritance MovieClip > UIObject class > ProgressBar

ActionScript Class Name mx.controls.ProgressBar

Setting a property of the ProgressBar class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.ProgressBar.version);
```

Note: The code `trace(myProgressBarInstance.version);` returns `undefined`.

Method summary for the ProgressBar class

The following table lists the method of the ProgressBar class.

Method	Description
`ProgressBar.setProgress()`	Sets the state of the progress bar to reflect the amount of progress made when the progress bar is in manual mode

Methods inherited from the UIObject class

The following table lists the methods the ProgressBar class inherits from the UIObject class. When calling these methods from the ProgressBar object, use the form `ProgressBar.`*`methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Property summary for the ProgressBar class

The following table lists properties of the ProgressBar class.

Property	Description
`ProgressBar.conversion`	A number used to convert the current bytes loaded value and the total bytes loaded values.
`ProgressBar.direction`	The direction in which the progress bar fills.

Property	Description
ProgressBar.indeterminate	Indicates whether the size of the loading source is unknown.
ProgressBar.label	The text that accompanies the progress bar.
ProgressBar.labelPlacement	The location of the label in relation to the progress bar.
ProgressBar.maximum	The maximum value of the progress bar in manual mode.
ProgressBar.minimum	The minimum value of the progress bar in manual mode.
ProgressBar.mode	The mode in which the progress bar loads content.
ProgressBar.percentComplete	Read-only; a number indicating the percent loaded.
ProgressBar.source	The content to load.
ProgressBar.value	Read-only; indicates the amount of progress that has been made.

Properties inherited from the UIObject class

The following table lists the properties the ProgressBar class inherits from the UIObject class. When calling these properties from the ProgressBar object, use the form `ProgressBar.propertyName`.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Event summary for the ProgressBar class

The following table lists events of the ProgressBar class.

Event	Description
`ProgressBar.complete`	Triggered when loading is complete.
`ProgressBar.progress`	Triggered as content loads in manual or polled mode.

Events inherited from the UIObject class

The following table lists the events the ProgressBar class inherits from the UIObject class. When calling these events from the ProgressBar object, use the form `ProgressBar.eventName`.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

ProgressBar.complete

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(complete){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.complete = function(eventObject){
   ...
}
pBar.addEventListener("complete", listenerObject)
```

Event object

In addition to the standard event object properties, there are two additional properties defined for the `ProgressBar.complete` event: `current` (the loaded value equals total), and `total` (the total value).

Description

Event; broadcast to all registered listeners when the loading progress has completed.

The first usage example uses an `on()` handler and must be attached directly to a ProgressBar instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the instance `pBar`, sends "_level0.pBar" to the Output panel:

```
on(complete){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`pBar`*) dispatches an event (in this case, `complete`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example creates a `form` listener object with a `complete` callback function that sends a message to the Output panel with the value of the `pBar` instance:

```
form.complete = function(eventObj){
  // eventObj.target is the component that generated the complete event,
  // i.e., the progress bar.
  trace("Current ProgressBar value = " + eventObj.target.value);
}
pBar.addEventListener("complete", form);
```

See also

`EventDispatcher.addEventListener()`

ProgressBar.conversion

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.conversion

Description

Property; a number that sets a conversion value for the incoming values. It divides the current and total values, floors them, and displays the converted value in the `label` property. The default value is 1.

Note: The floor is the closest integer value that is less than or equal to the specified value. For example, the number 4.6 becomes 4.

Example

The following code displays the value of the loading progress in kilobytes:

```
pBar.conversion = 1024;
```

ProgressBar.direction

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.direction

Description

Property; indicates the fill direction for the progress bar. The default value is `"right"`.

Example

The following code makes the progress bar fill from right to left:

```
pBar.direction = "left";
```

ProgressBar.indeterminate

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.indeterminate

Description

Property; a Boolean value that indicates whether the progress bar has a striped fill and a loading source of unknown size (`true`), or a solid fill and a loading source of a known size (`false`). For example, you might use this property if you are loading a large data set into a SWF file and do not know the size of the data you are loading.

Example

The following code creates a determinate progress bar with a solid fill that moves from left to right. Drag an instance of the ProgressBar component onto the Stage, and enter the instance name `my_pb` in the Property inspector. Drag an instance of the Loader component onto the Stage, and enter the instance name `my_ldr` in the Property inspector. Add the following code to Frame 1 of the Timeline:

```
var my_pb:mx.controls.ProgressBar;
var my_ldr:mx.controls.Loader;

var pbListener:Object = new Object();
pbListener.complete = function(evt:Object) {
  evt.target._visible = false;
};
my_pb.addEventListener("complete", pbListener);
my_pb.mode = "polled";
my_pb.indeterminate = true;
my_pb.source = my_ldr;

my_ldr.autoLoad = false;
my_ldr.scaleContent = false;
my_ldr.load("http://www.macromedia.com/software/flex/images/
  flex_presentation_eyes.jpg");
```

ProgressBar.label

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.label

Description

Property; text that indicates the loading progress. This property is a string in the format `"%1 out of %2 loaded (%3%%)"`. In this string, `%1` is a placeholder for the current bytes loaded, `%2` is a placeholder for the total bytes loaded, and `%3` is a placeholder for the percentage of content loaded. (The characters `%%` allow Flash to display a single `%` character.) If a value for `%2` is unknown, it is replaced by `??`. If a value is `undefined`, the label doesn't display. The default value is `"LOADING %3%%"`.

Example

The following code lets your application display progress bar text that reads "3 files loaded," "4 files loaded," and so on as the files load:

```
pBar.label = "%1 files loaded";
```

See also

`ProgressBar.labelPlacement`

ProgressBar.labelPlacement

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`pBarInstance``.labelPlacement`

Description

Property; sets the placement of the label in relation to the progress bar. The possible values are `"left"`, `"right"`, `"top"`, `"bottom"`, and `"center"`.

Example

The following code specifies that the text label appears above the progress bar:

```
pBar.label = "%1 out of %2 loaded (%3%%)";
pBar.labelPlacement = "top";
```

See also

`ProgressBar.label`

ProgressBar.maximum

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.maximum

Description

Property; the largest value for the progress bar when the ProgressBar.mode property is set to "manual".

Example

The following code sets the maximum property to the total frames of a Flash application that's loading:

```
pBar.maximum = _totalframes;
```

See also

ProgressBar.minimum, ProgressBar.mode

ProgressBar.minimum

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.minimum

Description

Property; the smallest value for the progress bar when the ProgressBar.mode property is set to "manual".

Example

The following code sets the minimum value for the progress bar:

```
pBar.minimum = 0;
```

See also

ProgressBar.maximum, ProgressBar.mode

ProgressBar.mode

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.mode

Description

Property; the mode in which the progress bar loads content. This value can be `"event"`, `"polled"`, or `"manual"`.

Event mode and polled mode are the most common modes. In event mode, the `source` property specifies loading content that emits `progress` and `complete` events; you should use a Loader object in this mode. In polled mode, the `source` property specifies loading content (such as a MovieClip object) that exposes `getBytesLoaded()` and `getsBytesTotal()` methods. Any object that exposes these methods can be used as a source in polled mode (including a custom object or the root Timeline).

You can also use the ProgressBar component in manual mode by manually setting the `maximum`, `minimum`, and `indeterminate` properties and making calls to the `ProgressBar.setProgress()` method.

Example

The following code sets the progress bar to event mode. Drag an instance of the ProgressBar component onto the Stage, and enter the instance name `my_pb` in the Property inspector. Drag an instance of the Loader component onto the Stage, and enter an instance name `my_ldr` in the Property inspector. Add the following code to Frame 1 of the Timeline:

```
var my_pb:mx.controls.ProgressBar;
var my_ldr:mx.controls.Loader;

var pbListener:Object = new Object();
pbListener.complete = function(evt:Object) {
  evt.target._visible = false;
};

my_pb.addEventListener("complete", pbListener);
my_pb.mode = "polled";
my_pb.indeterminate = true;
my_pb.source = my_ldr;

my_ldr.autoLoad = false;
my_ldr.scaleContent = false;
my_ldr.load("http://www.macromedia.com/software/flex/images/
  flex_presentation_eyes.jpg");
```

ProgressBar.percentComplete

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.percentComplete

Description

Property (read-only); tells what percentage of the content has been loaded. This value is floored. (The floor is the closest integer value that is less than or equal to the specified value. For example, the number 7.8 becomes 7.) The following formula is used to calculate the percentage:

```
100*(value-minimum)/(maximum-minimum)
```

Example

The following code sends the value of the `percentComplete` property to the Output panel:

```
trace("percent complete = " + pBar.percentComplete);
```

ProgressBar.progress

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(progress){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.progress = function(eventObject){
  ...
}
pBarInstance.addEventListener("progress", listenerObject)
```

Event object

In addition to the standard event object properties, there are two additional properties defined for the `ProgressBar.progress` event: `current` (the loaded value equals `total`), and `total` (the total value).

Description

Event; broadcast to all registered listeners whenever the value of a progress bar changes. This event is broadcast only when `ProgressBar.mode` is set to `"manual"` or `"polled"`.

The first usage example uses an `on()` handler and must be attached directly to a ProgressBar instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the instance `myPBar`, sends "_level0.myPBar" to the Output panel:

```
on(progress){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*pBarInstance*) dispatches an event (in this case, progress) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the EventDispatcher.addEventListener() method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example creates a listener object, form, and defines a progress event handler on it. The form listener is registered to the pBar instance in the last line of code. When the progress event is triggered, pBar broadcasts the event to the form listener, which calls the progress callback function.

```
var form:Object = new Object();
form.progress = function(eventObj){
  // eventObj.target is the component that generated the progress event,
  // i.e., the progress bar.
  trace("Current progress value = " + eventObj.target.value);
}
pBar.addEventListener("progress", form);
```

See also

EventDispatcher.addEventListener()

ProgressBar.setProgress()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.setProgress(*completed*, *total*)

Parameters

completed A number indicating the amount of progress that has been made. You can use the ProgressBar.label and ProgressBar.conversion properties to display the number in percentage form or any units you choose, depending on the source of the progress bar.

total A number indicating the total progress that must be made to reach 100%.

Returns

A number indicating the amount of progress that has been made.

Description

Method; sets the state of the progress bar to reflect the amount of progress made when the `ProgressBar.mode` property is set to `"manual"`. You can call this method to make the bar reflect the state of a process other than loading. For example, you might want to explicitly set the progress bar to zero progress.

The *`completed`* parameter is assigned to the `value` property and the *`total`* parameter is assigned to the `maximum` property. The `minimum` property is not altered.

Example

The following code calls `setProgress()` according to the progress of a Flash application's Timeline:

```
pBar.setProgress(_currentFrame, _totalFrames);
```

ProgressBar.source

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`pBarInstance``.source`

Description

Property; a reference to the instance to be loaded whose loading process will be displayed. The loading content should emit a `progress` event from which the current and total values are retrieved. This property is used only when `ProgressBar.mode` is set to `"event"` or `"polled"`. The default value is `undefined`.

The ProgressBar component can be used with content within an application, including `_root`.

Example

This example sets the `pBar` instance to display the loading progress of a Loader component with the instance name `loader`:

```
pBar.source = loader;
```

See also

`ProgressBar.mode`

ProgressBar.value

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

pBarInstance.value

Description

Property (read-only); indicates the amount of progress that has been made. This property is a number between the value of ProgressBar.minimum and ProgressBar.maximum. The default value is 0.

RadioButton component

The RadioButton component lets you force a user to make a single choice within a set of choices. This component must be used in a group of at least two RadioButton instances. Only one member of the group can be selected at any given time. Selecting one radio button in a group deselects the currently selected radio button in the group. You set the `groupName` parameter to indicate which group a radio button belongs to.

A radio button can be enabled or disabled. A disabled radio button doesn't receive mouse or keyboard input. When the user clicks or tabs into a RadioButton component group, only the selected radio button receives focus. The user can then use the following keys control it:

Key	Description
Up Arrow/Right Arrow	The selection moves to the previous radio button within the radio button group.
Down Arrow/ Left Arrow	The selection moves to the next radio button within the radio button group.
Tab	Moves focus from the radio button group to the next component.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each RadioButton instance on the Stage reflects changes made to parameters in the Property inspector or Component inspector during authoring. However, the mutual exclusion of selection does not display in the live preview. If you set the selected parameter to true for two radio buttons in the same group, they both appear selected even though only the last instance created will appear selected at runtime. For more information, see "RadioButton parameters" on page 622.

When you add the RadioButton component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code to enable accessibility:

```
mx.accessibility.RadioButtonAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the RadioButton component

A radio button is a fundamental part of any form or web application. You can use radio buttons wherever you want a user to make one choice from a group of options. For example, you would use radio buttons in a form to ask which credit card a customer wants to use.

RadioButton parameters

You can set the following authoring parameters for each RadioButton component instance in the Property inspector or in the Component inspector:

label sets the value of the text on the button; the default value is `Radio Button`.

data is the value associated with the radio button. There is no default value.

groupName is the group name of the radio button. The default value is `radioGroup`.

selected sets the initial value of the radio button to selected (`true`) or unselected (`false`). A selected radio button displays a dot inside it. Only one radio button in a group can have a selected value of `true`. If more than one radio button in a group is set to `true`, the radio button that is instantiated last is selected. The default value is `false`.

labelPlacement orients the label text on the button. This parameter can be one of four values: `left`, `right`, `top`, or `bottom`; the default value is `right`. For more information, see `RadioButton.labelPlacement`.

You can write ActionScript to set additional options for RadioButton instances using the methods, properties, and events of the RadioButton class. For more information, see "RadioButton class" on page 625.

Creating an application with the RadioButton component

The following procedure explains how to add RadioButton components to an application while authoring. In this example, the radio buttons are used to present the yes-or-no question "Are you a Flashist?". The data from the radio group is displayed in a TextArea component with the instance name `theVerdict`.

To create an application with the RadioButton component:

1. Drag two RadioButton components from the Components panel to the Stage.
2. Select one of the radio buttons. In the Component inspector, do the following:
   - Enter **Yes** for the label parameter.
   - Enter **Flashist** for the data parameter.
3. Select the other radio button. In the Component inspector, do the following:
   - Enter **No** for the label parameter.
   - Enter **Anti-Flashist** for the data parameter.
4. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
flashistListener = new Object();
flashistListener.click = function (evt){
    theVerdict.text = evt.target.selection.data
}
radioGroup.addEventListener("click", flashistListener);
```

   The last line of code adds a `click` event handler to the `radioGroup` radio button group. The handler sets the `text` property of `theVerdict` (a TextArea instance) to the value of the `data` property of the selected radio button in the `radioGroup` radio button group. For more information, see `RadioButton.click`.

Customizing the RadioButton component

You can transform a RadioButton component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`).

The bounding box of a RadioButton component is invisible and also designates the hit area for the component. If you increase the size of the component, you also increase the size of the hit area.

If the component's bounding box is too small to fit the component label, the label is clipped to fit.

Using styles with the RadioButton component

You can set style properties to change the appearance of a RadioButton. If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

A RadioButton component uses the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.

Style	Theme	Description
symbolBackgroundColor	Sample	The background color of the radio button. The default value is 0xFFFFFF (white).
symbolBackgroundDisabledColor	Sample	The background color of the radio button when disabled. The default value is 0xEFEEEF (light gray).
symbolBackgroundPressedColor	Sample	The background color of the radio button when pressed. The default value is 0xFFFFFF (white).
symbolColor	Sample	The color of the dot in the radio button. The default value is 0x000000 (black).
symbolDisabledColor	Sample	The color of the dot in the radio button when the component is disabled. The default value is 0x848384 (dark gray).

Using skins with the RadioButton component

You can skin the RadioButton component while authoring by modifying the component's symbols in the library. The skins for the RadioButton component are located in the following folder in the library of HaloTheme.fla or SampleTheme.fla: Flash UI Components 2/Themes/MMDefault/RadioButton Assets/States. See "About skinning components" on page 80.

If a radio button is enabled and unselected, it displays its rollover state when a user moves the pointer over it. When a user clicks an unselected radio button, the radio button receives input focus and displays its false pressed state. When a user releases the mouse, the radio button displays its true state and the previously selected radio button in the group returns to its false state. If a user moves the pointer off a radio button while pressing the mouse, the radio button's appearance returns to its false state and it retains input focus.

If a radio button or radio button group is disabled, it displays its disabled state, regardless of user interaction.

A RadioButton component uses the following skin properties:

Name	Description
falseUpIcon	The unselected state. The default value is RadioFalseUp.
falseDownIcon	The pressed-unselected state. The default value is RadioFalseDown.
falseOverIcon	The over-unselected state. The default value is RadioFalseOver.
falseDisabledIcon	The disabled-unselected state. The default value is RadioFalseDisabled.
trueUpIcon	The selected state. The default value is RadioTrueUp.
trueDisabledIcon	The disabled-selected state. The default value is RadioTrueDisabled.

Each of these skins correspond to the icon indicating the RadioButton state. The RadioButton does not have a border or background.

To create movie clip symbols for RadioButton skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the RadioButton Assets folder to the library for your document.
4. Expand the RadioButton Assets/States folder in the library of your document.
5. Open the symbols you want to customize for editing.

   For example, open the RadioFalseDisabled symbol.
6. Customize the symbol as desired.

   For example, change the inner white circle to a light gray.
7. Repeat steps 5-6 for all symbols you want to customize.

   For example, repeat the color change for the inner circle of the RadioTrueDisabled symbol.
8. Click the Back button to return to the main Timeline.
9. Drag a RadioButton component to the Stage.

   For this example, drag two instances to show the two new skin symbols.
10. Set the RadioButton instance properties as desired.

    For this example, set one RadioButton to selected, and use ActionScript to set both RadioButton instances to disabled.
11. Select Control > Test Movie.

RadioButton class

Inheritance MovieClip > UIObject class > UIComponent class > SimpleButton class > Button component > RadioButton

ActionScript Package Name mx.controls.RadioButton

The properties of the RadioButton class allow you at runtime to create a text label and position it in relation to the radio button. You can also assign data values to radio buttons, assign them to groups, and select them based on data value or instance name.

Setting a property of the RadioButton class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

The RadioButton component uses the Focus Manager to override the default Flash Player focus rectangle and draw a custom focus rectangle with rounded corners. For information about creating focus navigation, see "Creating custom focus navigation" on page 50.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.RadioButton.version);
```

Note: The code `trace(myRadioButtonInstance.version);` returns `undefined`.

Method summary for the RadioButton class

There are no methods exclusive to the RadioButton class.

Methods inherited from the UIObject class

The following table lists the methods the RadioButton class inherits from the UIObject class. When calling these methods from the RadioButton object, use the form *`RadioButtonInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the RadioButton class inherits from the UIComponent class. When calling these methods from the RadioButton object, use the form *`RadioButtonInstance.methodName`*.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Property summary for the RadioButton class

The following table lists properties of the RadioButton class.

Property	Description
`RadioButton.data`	The value associated with a radio button instance.
`RadioButton.groupName`	The group name for a radio button group instance or radio button instance.

Property	Description
RadioButton.label	The text that appears next to a radio button.
RadioButton.labelPlacement	The orientation of the label text in relation to a radio button or radio button group.
RadioButton.selected	Selects the radio button, and deselects the previously selected radio button. This property can be used with a RadioButton instance or a RadioButtonGroup instance.
RadioButton.selectedData	Selects the radio button with the specified data value in a radio button group.
RadioButton.selection	A reference to the currently selected radio button in a radio button group. This property can be used with a RadioButton instance or a RadioButtonGroup instance.

Properties inherited from the UIObject class

The following table lists the properties the RadioButton class inherits from the UIObject class. When accessing these properties from the RadioButton object, use the form *RadioButtonInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the RadioButton class inherits from the UIComponent class. When accessing these properties from the RadioButton object, use the form *RadioButtonInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the SimpleButton class

The following table lists the properties RadioButton class inherits from the SimpleButton class. When accessing these properties from the RadioButton object, use the form *RadioButtonInstance.propertyName*.

Property	Description
SimpleButton.emphasized	Indicates whether a button has the appearance of a default push button.
SimpleButton.emphasizedStyleDeclaration	The style declaration when the `emphasized` property is set to `true`.
SimpleButton.selected	A Boolean value indicating whether the button is selected (`true`) or not (`false`). The default value is `false`.
SimpleButton.toggle	A Boolean value indicating whether the button behaves as a toggle switch (`true`) or not (`false`). The default value is `false`.

Properties inherited from the Button class

The following table lists the properties the RadioButton class inherits from the Button class. When accessing these properties from the RadioButton object, use the form *RadioButtonInstance.propertyName*.

Property	Description
Button.icon	Specifies an icon for a button instance.
Button.label	Specifies the text that appears in a button.
Button.labelPlacement	Specifies the orientation of the label text in relation to an icon.

Event summary for the RadioButton class

The following table lists the event of the RadioButton class.

Event	Description
RadioButton.click	Triggered when the mouse is clicked over a radio button or radio button group.

Events inherited from the UIObject class

The following table lists the events the RadioButton class inherits from the UIObject class.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the RadioButton class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

Events inherited from the SimpleButton class

The following table lists the event the RadioButton class inherits from the SimpleButton class.

Event	Description
`SimpleButton.click`	Broadcast when the mouse is clicked (released) over a button or if the button has focus and the Spacebar is pressed.

RadioButton.click

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(click){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.click = function(eventObject){
  ...
}
radioButtonGroup.addEventListener("click", listenerObject)
```

Description

Event; broadcast to all registered listeners when the mouse is clicked (pressed and released) over the radio button or if the radio button is selected by means of the arrow keys. The event is also broadcast if the Spacebar or arrow keys are pressed when a radio button group has focus, but none of the radio buttons in the group are selected.

The first usage example uses an `on()` handler and must be attached directly to a RadioButton instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the radio button `myRadioButton`, sends "_level0.myRadioButton" to the Output panel:

```
on(click){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`radioButtonInstance`*) dispatches an event (in this case, `click`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. The event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a radio button in `radioGroup` is clicked. The first line of code creates a listener object called `form`. The second line defines a function for the `click` event on the listener object. Inside the function is a `trace()` statement that uses the event object (`eventObj`) that is automatically passed to the function to generate a message. The `target` property of an event object is the component that generated the event. You can access instance properties from the `target` property (in this example, the `RadioButton.selection` property is accessed). The last line calls `EventDispatcher.addEventListener()` from `radioGroup` and passes it the `click` event and the `form` listener object as parameters.

```
form = new Object();
form.click = function(eventObj){
  trace("The selected radio instance is " + eventObj.target.selection);
```

```
}
radioGroup.addEventListener("click", form);
```

The following code also sends a message to the Output panel when `radioButtonInstance` is clicked. The `on()` handler must be attached directly to `radioButtonInstance`.

```
on(click){
  trace("radio button component was clicked");
}
```

RadioButton.data

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`radioButtonInstance``.data`

Description

Property; specifies the data to associate with a RadioButton instance. Setting this property overrides the data parameter value set during authoring. The `data` property can be of any data type.

Example

The following example assigns the data value `"#FF00FF"` to the `radioOne` radio button instance:

```
radioOne.data = "#FF00FF";
```

RadioButton.groupName

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`radioButtonInstance``.groupName`

`radioButtonGroup``.groupName`

Description

Property; sets the group name for a radio button instance or group. You can use this property to get or set a group name for a radio button instance or for a radio button group. Calling this method overrides the groupName parameter value set during authoring. The default value is `"radioGroup"`.

Example

The following example sets the group name of a radio button instance to `colorChoice` and then changes the group name to `sizeChoice`. To test this example, place a radio button on the Stage, name the instance name `myRadioButton`, and enter the following code on Frame 1:

```
myRadioButton.groupName = "colorChoice";
trace(myRadioButton.groupName);
colorChoice.groupName = "sizeChoice";
trace(colorChoice.groupName);
```

RadioButton.label

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

radioButtonInstance.label

Description

Property; specifies the text label for the radio button. By default, the label appears to the right of the radio button. Calling this method overrides the label parameter specified during authoring. If the label text is too long to fit within the bounding box of the component, the text is clipped.

Example

The following example sets the `label` property of the instance `radioButton`:

```
radioButton.label = "Remove from list";
```

RadioButton.labelPlacement

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

radioButtonInstance.labelPlacement

radioButtonGroup.labelPlacement

Description

Property; a string that indicates the position of the label in relation to a radio button. You can set this property for an individual instance or for a radio button group. If you set the property for a group, the label is placed in the appropriate position for each radio button in the group.

The following are the four possible values:

- `"right"` The radio button is pinned to the upper left corner of the bounding area. The label is placed to the right of the radio button.
- `"left"` The radio button is pinned to the upper right corner of the bounding area. The label is placed to the left of the radio button.
- `"bottom"` The label is placed below the radio button. The radio button and label grouping are centered horizontally and vertically. If the bounding box of the radio button isn't large enough, the label is clipped.
- `"top"` The label is placed above the radio button. The radio button and label grouping are centered horizontally and vertically. If the bounding box of the radio button isn't large enough, the label is clipped.

Example

The following code places the label to the left of each radio button in `radioGroup`:

```
radioGroup.labelPlacement = "left";
```

RadioButton.selected

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`radioButtonInstance``.selected`

`radioButtonGroup``.selected`

Description

Property; a Boolean value that sets the state of the radio button to selected (`true`) and deselects the previously selected radio button, or sets the radio button to deselected (`false`).

Example

The first line of code sets the `mcButton` instance to `true`. The second line of code returns the value of the `selected` property.

```
mcButton.selected = true;
trace(mcButton.selected);
```

RadioButton.selectedData

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

radioButtonGroup.selectedData

Description

Property; selects the radio button with the specified data value and deselects the previously selected radio button. If the `data` property is not specified for a selected instance, the `label` value of the selected instance is selected and returned. The `selectedData` property can be of any data type.

Example

The following example selects the radio button with the value `"#FF00FF"` from the radio group `colorGroup` and sends the value to the Output panel:

```
colorGroup.selectedData = "#FF00FF";
trace(colorGroup.selectedData);
```

RadioButton.selection

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

radioButtonInstance.selection

radioButtonGroup.selection

Description

Property; behaves differently depending on whether you get or set the property. If you get the property, it returns the object reference of the currently selected radio button in a radio button group. If you set the property, it selects the specified radio button (passed as an object reference) in a radio button group and deselects the previously selected radio button.

Example

The following example selects the radio button with the instance name `color1` and sends its instance name to the Output panel:

```
colorGroup.selection = color1;
trace(colorGroup.selection._name)
```

RadioButtonGroup component

For information about the RadioButtonGroup class, see RadioButton component.

RDBMSResolver component (Flash Professional only)

Resolver components are used with the DataSet component (part of the data management functionality in the Flash data architecture) to save changes to an external data source. Resolvers include both the RDBMSResolver component and the XUpdateResolver component. Resolvers take a delta packet (returned by `DataSet.deltaPacket`) and convert it to an update packet in a format appropriate to the type of resolver. The update packet can then be transmitted to the external data source by one of the connector components. Resolver components have no visual appearance at runtime.

The RDBMSResolver component creates an XML update packet that can be easily parsed by into SQL statements for updating a relational database. The RDBMSResolver component is connected to a DataSet component's DeltaPacket property, sends its own update packet to a connector, receives server errors back from the connector, and communicates them back to the DataSet component—all using bindable properties.

For more information about the Flash data architecture, see "Data resolution (Flash Professional only)" in *Using Flash*. For more information about relational databases, see "Resolving data to a relational database (Flash Professional only)" in *Using Flash*. For a complete example of an application that updates data using the RDBMSResolver component, see www.macromedia.com/devnet/mx/flash/articles/delta_packet.html.

Note: You can use the RDBMSResolver component to send data updates to any object you write that can parse XML and generate SQL statements against a database—for example, an ASP page, a Java servlet, or a ColdFusion component.

Using the RDBMSResolver component (Flash Professional only)

You use the RDBMSResolver component only when your Flash application contains a DataSet component and must send an update back to the data source. This component resolves data that you want to return to a relational database.

RDBMSResolver parameters

You can set the following authoring parameters for each RDBMSResolver instance by using the Parameters tab of the Component inspector:

TableName is a string representing **the name (in the XML) of the** database table to be updated. This string should match the name of the `RDBMSResolver.fieldInfo` item to be updated. If there are no updates to this field, this parameter should be blank, which is the default value.

UpdateMode is an enumerator that determines the way key fields are identified when the XML update packet is generated. Possible values are as follows:

- `umUsingAll` Uses the old values of all of the modified fields to identify the record to be updated. This is the safest value to use for updating, because it guarantees that another user has not modified the record since you retrieved it. However, this approach is time consuming and generates a larger update packet.
- `umUsingModified` Uses the old values of all of the fields modified to identify the record to be updated. This value guarantees that another user has not modified the same fields in the record since you retrieved it.

- `umUsingKey` The default value. This setting uses the old value of the key fields. This implies an "optimistic concurrency" model, which most database systems today employ, and guarantees that you are modifying the same record that you retrieved from the database. Your changes overwrites any other user's changes to the same data.

NullValue is a string representing a null field value. You can customize this parameter to prevent it from being confused with an empty string (`""`) or another valid value. The default value is `{_NULL_}`.

FieldInfo is a collection representing one or more key fields that uniquely identify the records. If your data source is a database table, the table should have one or more fields that uniquely key the records within it. Additionally, some fields may have been calculated or joined from other tables. Those fields must be identified so that the key fields can be set within the XML update packet, and so that any fields that should not be updated are omitted from the XML update packet.

The FieldInfo parameter lets you use properties to designate fields that require special handling. Each item in the collection contains three properties:

- `FieldName` Name of a field. This should match a field name in the DataSet component.
- `OwnerName` Optional value used to identify fields not "owned" by the same table defined in the RDBMSResolver component's TableName parameter. If this property has the same value as the TableName parameter or is blank, usually the field is included in the XML update packet. If it has a different value, this field is excluded from the update packet.
- `IsKey` Boolean property that you should set to `true` so that all key fields for the table are updated.

The following example shows FieldInfo items that are created to update fields in a customer table. You must identify the key fields in the customer table. The customer table has a single key field, `id`; therefore, you should create a field item with the following values:

```
FieldName = "id"
OwnerName = <--! leave this value blank -->
IsKey = "true"
```

Also, the `custType` field is added by means of a join in the query. Because this field should be excluded from the update, you create a field item with the following values:

```
FieldName = "custType"
OwnerName = "JoinedField"
IsKey = "false"
```

When the field items are defined, Flash Player can use them to automatically generate the complete XML, which is used to update a table.

Note: The FieldInfo parameter makes use of a Flash feature called the Collection Editor. When you select the FieldInfo parameter, you can use the Collection Editor dialog box to add new FieldInfo items and set their `fieldName`, `ownerName`, and `isKey` properties from one location.

Common workflow for the RDBMSResolver component

The following steps describe the typical workflow for the RDBMSResolver component.

To use an RDBMSResolver component:

1. Add two instances of the WebServiceConnector component and one instance of the DataSet and RDBMSResolver components to your application, and give them instance names.
2. Select the first WebServiceConnector component. Then use the Parameters tab of the Component inspector to enter the Web Service Definition Language (WSDL) URL for a web service that exposes data from an external data source.

   Note: The web service must return an array of records to be bound to the data set.
3. Use the Bindings tab of the Component inspector to bind the first WebServiceConnector component's results property to the DataSet component's `dataProvider` property.
4. Select the DataSet component, and use the Bindings tab of the Component inspector to bind data elements (DataSet fields) to the visual components in your application.
5. Bind the DataSet's deltaPacket property to the RDBMSResolver's deltaPacket property.

   The update instructions are sent from the DataSet component to the RDBMSResolver component when the `DataSet.applyUpdates()` method is called.
6. Bind the RDBMSResolver's updatePacket property to the second WebServiceConnector's params property to send data back to a method that will parse the XML update packet. Set the kind of that params property to auto-trigger so that the connector will send the update packet as soon as data binding copies it over.
7. Add a trigger to initiate the data binding operation: use the Trigger Data Source behavior attached to a button, or add ActionScript.

In addition to these steps, you can also use the RDBMSResolver component to create bindings to apply the result packet sent back from the server to the data set.

www.macromedia.com/devnet/mx/flash/ data_integration.html.

RDBMSResolver class (Flash Professional only)

Inheritance MovieClip > RDBMSResolver

ActionScript Package Name mx.data.components.RDBMSResolver

The methods, properties, and events of the RDBMSResolver class allow you to connect to a DataSet component and make changes to external data sources.

Method summary for the RDBMSResolver component

The following table lists the method of the RDBMSResolver class.

Method	Description
`RDBMSResolver.addFieldInfo()`	Adds a new item to the `fieldInfo` collection, which is used for setting up an RDBMSResolver component dynamically at runtime.

Property summary for the RDBMSResolver component

The following table lists properties of the RDBMSResolver class.

Property	Description
`RDBMSResolver.deltaPacket`	The DataSet object's `deltaPacket` property should be bound to this property so that when `DataSet.applyUpdates()` is called, the binding will copy it across and the resolver will create the update packet.
`RDBMSResolver.fieldInfo`	A collection of fields with properties that identify DataSet fields that require special handling, either because they are key fields or because they cannot be updated.
`RDBMSResolver.nullValue`	A string that is placed in the update packet to indicate that a field's value is `null`.
`RDBMSResolver.tableName`	Identifies the database table that is to be updated.
`RDBMSResolver.updateMode`	Values that determine how key fields are identified when the XML update packet is generated.
`RDBMSResolver.updatePacket`	The XML packet produced by this resolver that contains the changes from the data set's delta packet.
`RDBMSResolver.updateResults`	A delta packet that contains the results of an update returned from the server through a connector.

Event summary for the RDBMSResolver component

The following table lists the events of the RDBMSResolver class.

Event	Description
`RDBMSResolver.beforeApplyUpdates`	Defined in your application; called by the RDBMSResolver component to make custom modifications to the XML of the `updatePacket` property before it is bound to the connector.
`RDBMSResolver.reconcileResults`	Defined in your application; called by the RDBMSResolver component to compare two packets after results have been received from the server and applied to the delta packet.
`RDBMSResolver.reconcileUpdates`	Defined in your application; called by the RDBMSResolver component when results have been received from the server after the updates from a delta packet were applied.

RDBMSResolver.addFieldInfo()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
resolveData.addFieldInfo("fieldName", "ownerName", "isKey")
```

Parameters

fieldName String; provides the name of the field that this information object describes.

ownerName String; provides the name of the table that owns this field. If this name is the same as the RDBMSResolver instance's `tableName` property, you can leave this parameter blank (`""`).

isKey Boolean; indicates whether this field is a key field.

Returns

Nothing.

Description

Method; adds a new item to the XML `fieldInfo` collection in the update packet. Use this method if you must set up an RDBMSResolver component dynamically at runtime, rather than using the Component inspector in the authoring environment.

Example

The following example creates an RDBMSResolver component and provides the name of the table, provides the name of the key field, and prevents the `personTypeName` field from being updated:

```
var myResolver:RDBMSResolver = new RDBMSResolver();
myResolver.tableName = "Customers";
// Sets up the id field as a key field
// and the personTypeName field so it won't be updated.
myResolver.addFieldInfo("id", "", true);
myResolver.addFieldInfo("personTypeName", "JoinedField", false);
// Sets up the data bindings
//...
```

RDBMSResolver.beforeApplyUpdates

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData`.beforeApplyUpdates(`*eventObject*`)`

Parameters

eventObject Resolver event object; describes the customizations to the XML packet before the update is sent though the connector to the database. This event object should contain the following properties:

Property	Description
`target`	Object; the resolver producing this event.

Property	Description
`type`	String; the name of the event.
`updatePacket`	XML object; the XML object about to be applied.

Returns

Nothing.

Description

Property; a property of type `deltaPacket`. It receives a delta packet to be translated into an update packet, and outputs a delta packet from any server results placed in the `updateResults` property. This event handler provides a way for you to make custom modifications to the XML before sending the updated data to a connector.

Messages in the `updateResults` property are treated as errors. This means that a delta with messages is added to the delta packet again so it can be re-sent the next time the delta packet is sent to the server. You must write code to handle deltas that have messages so that the messages are presented to the user and the deltas can be modified before being added to the next delta packet.

Example

The following example adds the user authentication data to the XML packet:

```
on (beforeApplyUpdates) {
   // add user authentication data
   var userInfo = new XML(""+getUserId()+ ""+getPassword()+"");
   updatePacket.firstChild.appendChild(userInfo);
}
```

RDBMSResolver.deltaPacket

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.deltaPacket`

Description

Property; a property of type `deltaPacket`. It receives a delta packet to be translated into an update packet, and outputs a delta packet from any server results placed in the `updateResults` property.

Messages in the `updateResults` property are treated as errors. This means that a delta with messages is added to the delta packet again so it can be re-sent the next time the delta packet is sent to the server. You must write code to handle deltas that have messages so that the messages are presented to the user and the deltas can be modified before being added to the next delta packet.

RDBMSResolver.fieldInfo

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.fieldInfo`

Description

Property; specifies a collection of an unlimited number of fields with properties that identify DataSet fields that require special handling, either because they are key fields or because they cannot be updated. Each `fieldInfo` item in the collection contains three properties:

Property	Description
`FieldName`	Name of the special-case field. This field name should match a field name in the DataSet.
`OwnerName`	An optional property. If this field is not "owned" by the table defined in the `RDBMSResolver.tableName` property, `OwnerName` is the name of the owner of this field. If `OwnerName` has the same value as `RDBMSResolver.tableName` or is blank, usually the field is included in the XML update packet. If `OwnerName` doesn't have any of these values, this field is excluded from the update packet.
`IsKey`	A Boolean value; if `true`, all key fields for the table are updated.

RDBMSResolver.nullValue

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.nullValue`

Description

Property; a string used to provide a null value for a field's value. You can customize this property to prevent it from being confused with an empty string (`""`) or another valid value. The default string is `{_NULL_}`.

RDBMSResolver.reconcileResults

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.reconcileResults(*eventObject*)

Parameters

eventObject Resolver event object; describes the event object used to compare two update packets. This event object should contain the following properties:

Property	Description
target	Object; the resolver broadcasting this event.
type	String; the name of the event.

Returns

Nothing.

Description

Event; broadcast by the RDBMSResolver component to compare two packets after results have been received from the server and applied to the delta packet.

A single updateResults packet can contain results of operations that were in the delta packet, as well as information about updates performed by other clients. When a new update packet is received, the operation results and database updates are split into two update packets and placed separately in the deltaPacket property. The reconcileResults event is broadcast just before the delta packet containing the operation results is sent using data binding.

Example

The following example reconciles two update packets and returns and clears the updates on success:

```
on (reconcileResults) {
      // examine results
      if(examine(updateResults))
        myDataSet.purgeUpdates();
      else
       displayErrors(results);
}
```

RDBMSResolver.reconcileUpdates

Availability

Flash Player 7.

Edition

This method is not currently available; for more information, see Flash MX 2004 release notes.

Usage

```
resolveData.reconcileUpdates(eventObject)
```

Parameters

eventObject Resolver event object; describes the customizations to the XML packet before the update is sent through the connector to the database. This event object should contain the following properties:

Property	Description
`target`	Object; the resolver broadcasting this event.
`type`	String; the name of the event.

Returns

None.

Description

This method is currently not available; for more information, see Flash MX 2004 release notes. A workaround for this method is to load or refresh the data set using a connector component and write ActionScript code to translate the field values you are trying to reconcile.

Event; broadcast by the RDBMSResolver component when results have been received from the server after the updates from a delta packet were applied. A single `updateResults` packet can contain results of operations that were in the delta packet, as well as information about updates that were performed by other clients. When a new update packet is received, the operation results and database updates are split into two delta packets, which are placed separately in the `deltaPacket` property. The `reconcileUpdates` event is broadcast just before the delta packet containing any database updates is sent using data binding.

Example

The following example reconciles two results and clears the updates on success:

```
on (reconcileUpdates) {
      // examine results
      if(examine(updateResults))
        myDataSet.purgeUpdates();
      else
       displayErrors(results);
```

RDBMSResolver.tableName

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.tableName

Description

Property; a string that represents the table name in the XML for the database table to be updated. This property also determines which fields to send in the update packet. To make this determination, the RDBMSResolver component compares the value of this property with the value provided for the `fieldInfo.ownerName` property. If a field has no entry in the `fieldInfo` collection property, the field is placed in the update packet. If a field has an entry in the `fieldInfo` collection property, and its `ownerName` value is blank or identical to the RDBMSResolver component's `tableName` property, the field is placed in the update packet. If a field has an entry in the `fieldInfo` collection property, and its `ownerName` value is not blank and is different from the RDBMSResolver component's `tableName` property, the field is not placed in the update packet.

RDBMSResolver.updateMode

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.updateMode

Description

Property; contains several values that determine how key fields are identified when the XML update packet is generated. This property can have the following strings as values:

Value	Description
`"umUsingAll"`	Uses the old values of all of the modified fields to identify the record to be updated. This is the safest value to use for updating, because it guarantees that another user has not modified any field of the record since you retrieved it. However, this approach is more time consuming and generates a larger update packet.
`"umUsingModified"`	Uses the old values of all of the fields modified to identify the record to be updated. This value guarantees that another user has not modified the same fields in the record since you retrieved it.
`"umUsingKey"`	The default value. This setting uses the old value of the key fields. This implies an "optimistic concurrency" model, which most database systems today employ, and guarantees that you are modifying the same record that you retrieved from the database. Your changes overwrite any other user's changes to the same data.

RDBMSResolver.updatePacket

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.updatePacket

Description

Property; property of type XML, containing an XML packet used to bind to a connector property that transmits the translated update packet of changes back to the server so the source of the data can be updated. This is an XML document containing the packet of DataSet changes.

RDBMSResolver.updateResults

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.updateResults

Description

Property; a delta packet that contains the results of an update returned from the server through a connector. Use this property to transmit errors and updated data from the server to a data set—for example, when the server assigns new IDs for an auto-assigned field. Bind this property to a connector's `results` property so that it can receive the results of an update and transmit the results back to the data set.

Messages in the `updateResults` property are treated as errors. This means that a delta with messages is added to the delta packet again so it can be re-sent the next time the delta packet is sent to the server. You must write code to handle deltas that have messages so that the messages are presented to the user and the deltas can be modified before being added to the next delta packet.

RectBorder class

The RectBorder class is used as the border of most components. A separate implementation of this class is provided by each theme, which has its own set of border styles and properties that it supports.

You interact with the RectBorder class primarily by setting styles on other components. For example, when you include a List component in a document and set the `borderStyle` style property, the List component creates a RectBorder instance that uses the list's `borderStyle` setting. You may also create a custom RectBorder implementation to skin the border of all components that use RectBorder.

The RectBorder class has four standard display styles: `none`, `inset`, `outset`, and `solid`.

The Halo theme also adds four special display styles, which are used by specific components.

Special style	Component that uses it
`default`	Window
`alert`	Alert
`dropDown`	ComboBox and DateField
`menuBorder`	Menu and MenuBar

The RectBorder behavior and style properties described here are consistent for all components that utilize the RectBorder class.

Using styles with the RectBorder class

You can set style properties to change the appearance of a RectBorder instance. A RectBorder instance uses the following styles:

- `borderCapColor`
- `borderColor`
- `buttonColor`
- `highlightColor`
- `shadowCapColor`
- `shadowColor`
- `themeColor`

The styles available on a particular RectBorder instance depend on the theme in use and the border style set on the component. For an interactive demonstration that shows the relationship between theme, border style, and available color style properties, see Using Components Help.

The four special Halo styles—`default`, `alert`, `dropDown`, and `menuBorder`—have some lines whose colors cannot be set through styles. You can only modify these colors by creating a custom theme and modifying the appropriate ActionScript within the custom RectBorder implementation.

Creating a custom RectBorder implementation

The RectBorder class is used as a border skin in most version 2 components. The default implementations in both the Halo and Sample themes use ActionScript to draw the border. A custom implementation must use ActionScript to register itself as the RectBorder implementation and provide sizing functionality, but can use either ActionScript or graphic elements to represent the visuals.

Each RectBorder implementation must adhere to the following requirements:

- It must extend mx.skins.RectBorder or one of its subclasses.
- It must provide an `offset` property value or implement the `getBorderMetrics` method to return sizing information.
- It must implement the `drawBorder()` method to draw or size the border.
- It must support all four standard styles, as well as the four special styles.

  The implementation can reuse standard borders for special borders, as the Sample theme does.
- It must register itself as the RectBorder implementation.

RectBorder global registration

All components look to a central location for a reference to the RectBorder class in use for the document, `_global.styles.rectBorderClass`. You cannot specify that an individual component should use a different RectBorder implementation. To customize RectBorder for a component, you must rely on the `borderStyle` style property.

A custom RectBorder example

Both RectBorder implementations provided by the Halo theme and the Sample theme use the ActionScript drawing API to draw the borders for different styles. The following example demonstrates how to create a custom RectBorder implementation that utilizes graphic symbols for its display.

To create a custom RectBorder implementation:

1. Create a new folder in the Classes/mx/skins folder corresponding to the custom package name you'll use for the custom border.

   For this example, use **myTheme**.
2. Create a new AS file in the new folder, and save it as RectBorder.as.
3. Copy the following ActionScript code to the new AS file:

```
import mx.core.ext.UIObjectExtensions;

class mx.skins.myTheme.RectBorder extends mx.skins.RectBorder
{
  static var symbolName:String = "RectBorder";
  static var symbolOwner:Object = RectBorder;
  var className:String = "RectBorder";

#include "../../core/ComponentVersion.as"
```

```
    // all of these borders have the same size edges, one pixel
    var offset:Number = 1;

    function init(Void):Void
    {
      super.init();
    }

    function drawBorder(Void):Void
    {
       // the graphics are on the symbol's Timeline,
       // so all you need to do here is size the border
       _width = __width;
       _height = __height;
    }

    // register ourselves as the RectBorder for all components to use
    static function classConstruct():Boolean
    {
      UIObjectExtensions.Extensions();
      _global.styles.rectBorderClass = RectBorder;
      _global.skinRegistry["RectBorder"] = true;
      return true;
    }
    static var classConstructed:Boolean = classConstruct();
    static var UIObjectExtensionsDependency = UIObjectExtensions;
}
```

If you're not using the package myTheme, change the class declaration as needed.

4. Save the AS file.
5. Create a new FLA file.
6. Use Insert > New Symbol to create a new movie clip symbol.
7. Set the name to `RectBorder`.
8. If the advanced fields are not displayed, click Advanced.
9. Select Export for ActionScript

   The identifier will be automatically filled in as `RectBorder`.
10. Set the AS 2.0 class to the full class name of the custom border implementation.

    This example uses mx.skins.myTheme.RectBorder.
11. Ensure that Export in First Frame is selected, and click OK.
12. Open the RectBorder symbol for editing.
13. Draw the graphics for the symbol.

    For example, draw a hairline square with no fill. To make the custom border easy to see, set the line color to bright red.

14. Ensure that the graphics are flush against the upper left corner with *x* and *y* coordinates set to (0,0).

    Your custom `drawBorder` implementation will set the width and height according to the component requirements.

15. Click Back to return to the main Timeline.

16. Drag several components that use RectBorder to the Stage.

    For example, drag a List, TextArea, and TextInput component to the Stage.

17. Select Control > Test Movie.

The above example creates a very simple border implementation with static color and graphics and doesn't respond to different `borderStyle` settings; it always uses the same graphics regardless of `borderStyle`. For an example of a more complete border implementation, review those provided by the Halo and Sample themes.

Screen class (Flash Professional only)

The Screen class is the base class for screens you create in the Screen Outline pane in Flash MX Professional 2004. Screens are high-level containers for creating applications and presentations. For an overview of working with screens, see Chapter 12, "Working with Screens (Flash Professional Only)," in *Using Flash*.

The Screen class has two primary subclasses: Slide and Form.

The Slide class provides the runtime behavior for slide presentations. The Slide class provides built-in navigation and sequencing capabilities, and lets you easily attach transitions between slides using behaviors. Slide objects maintain "state," and allow the user to advance to the next or previous slide/state: when the next slide is shown, the previous slide is hidden. For more information about using the Slide class to control slide presentations, see "Slide class (Flash Professional only)" on page 693.

The Form class provides the runtime environment for form applications. Forms can overlay and contain, or be contained by, other components. Unlike slides, forms don't provide any sequencing or navigation capabilities. For more information, see "Form class (Flash Professional only)" on page 430.

The Screen class provides functionality common to both slides and forms.

Screens know how to manage their children Every screen includes a built-in property that contains a list of that screen's child screens, known as a collection. This collection is determined by the screen hierarchy in the Screen Outline pane. Screens can have any number of children (or none), which themselves can have children.

Screens can hide and show their children Because a screen is, essentially, a collection of nested movie clips, a screen can control the visibility of its children. For form applications, all of a screen's children are visible by default at the same time; for slide presentations, individual screens are typically shown one at a time.

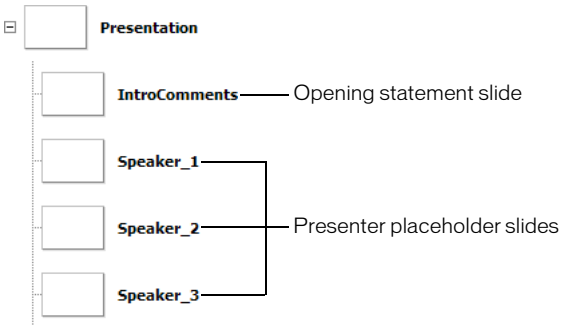
Screens broadcast events You can, for example, trigger a sound to play, or start playing some video, when a particular screen becomes visible.

Loading external content into screens (Flash Professional only)

The Screen class extends the Loader class (see "Loader component" on page 484), which lets you easily manage and load external SWF and JPEG files. The Loader class contains a `contentPath` property, which specifies the URL of an external SWF or JPEG file, or the linkage identifier of a movie clip in the library.

Using this feature, you can load an external screen tree (or any external SWF file) as a child of any screen node. This provides a useful way to make your screen-based media modular and divide it into separate SWF files.

For example, suppose you have a slide presentation in which three people are each contributing a single section. You could ask each presenter to create a separate slide presentation (SWF file). You would then create a "master slide presentation" that contains three placeholder slides, one for each slide presentation being created by the presenters. For each placeholder slide, you could point its `contentPath` property to one of the SWF files. The master slide presentation could be arranged as shown in the following illustration:



"Master" SWF slide presentation structure

Suppose presenters provide you with three SWF files, speaker_1.swf, speaker_2.swf, and speaker_3.swf. You could easily assemble the overall presentation by setting the `contentPath` property of each placeholder slide, either using the Property inspector or ActionScript, as shown in the following code:

```
Speaker_1.contentPath = speaker_1.swf;
Speaker_2.contentPath = speaker_2.swf;
Speaker_3.contentPath = speaker_3.swf;
```

By default, when you set a slide's `contentPath` property while authoring in the Property inspector, or using ActionScript (as shown above), the specified SWF file loads as soon as the "master presentation" SWF file has loaded. To reduce initial load time, consider setting the `contentPath` property in an `on(reveal)` handler attached to each slide.

```
// Attached to Speaker_1 slide
on(reveal) {
  this.contentPath="speaker_1.swf";
}
```

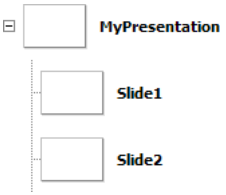
Alternatively, you could set the slide's `autoLoad` property to `false`. Then you could call the `load()` method on the slide when the slide has been revealed. (The `autoLoad` property and the `load()` method are inherited from the Loader class.)

```
// Attached to Speaker_1 slide
on(reveal) {
  this.load();
}
```

Referencing loaded screens with ActionScript

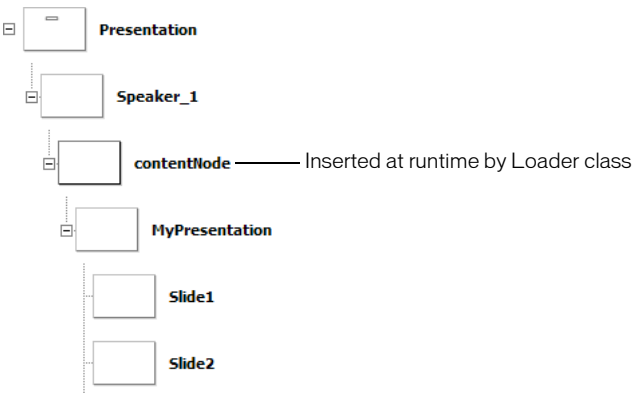
The Loader class creates an internal movie clip named `contentNode` into which it loads the SWF or JPEG file specified by the `contentPath` property. This movie clip, in effect, adds an extra screen node between the "placeholder" slide (that you created in the "master" presentation above) and the first slide in the loaded slide presentation.

For example, suppose the SWF file created for the Speaker_1 slide placeholder (see above illustration) had the following structure, as shown in the Screen Outline pane:



"Speaker 1" SWF slide presentation structure

At runtime, when the Speaker 1 SWF file is loaded into the placeholder slide, the overall slide presentation would have the following structure:



Structure of "master" and "speaker" presentation (runtime)

The properties and methods of the Screen, Slide, and Form classes "ignore" this `contentHolder` node as much as possible. That is, the slide named MyPresentation (along with its subslides) is part of the contiguous slide tree rooted at the Presentation slide, and is not treated as a separate subtree.

Screen class (API) (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > Loader component > Screen

ActionScript Class Name mx.screens.Screen

The methods, properties, and events of the Screen class allow you to create and manipulate screens at runtime.

Method summary for the Screen class

The following table lists the method of the Screen class.

Method	Description
`Screen.getChildScreen()`	Returns the child screen of this screen at a particular index.

Methods inherited from the UIObject class

The following table lists the methods the Screen class inherits from the UIObject class. When calling these methods from the Screen object, use the form *`ScreenInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Screen class inherits from the UIComponent class. When calling these methods from the Screen object, use the form *`ScreenInstance.methodName`*.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Methods inherited from the Loader class

The following table lists the method the Screen class inherits from the Loader class. When calling this method from the Screen object, use the form *ScreenInstance.methodName*.

Method	Description
Loader.load()	Loads the content specified by the contentPath property.

Property summary for the Screen class

The following table lists properties of the Screen class.

Property	Description
Screen.currentFocusedScreen	Read-only; returns the screen that contains the global current focus.
Screen.indexInParent	Read-only; returns the screen's index (zero-based) in its parent screen's list of child screens.
Screen.numChildScreens	Read-only; returns the number of child screens contained by the screen.
Screen.parentIsScreen	Read-only; returns a Boolean (true or false) value that indicates whether the screen's parent object is itself a screen.
Screen.parentScreen	Read-only; returns the screen that contains the specified screen.
Screen.rootScreen	Read-only; returns the root screen of the tree or subtree that contains the screen.

Properties inherited from the UIObject class

The following table lists the properties the Screen class inherits from the UIObject class. When accessing these properties from the Screen object, use the form *ScreenInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.

Property	Description
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Screen class inherits from the UIComponent class. When accessing these properties from the Screen object, use the form *ScreenInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the Loader class

The following table lists the properties the Screen class inherits from the Loader class. When accessing these properties from the Screen object, use the form *ScreenInstance.propertyName*.

Property	Description
Loader.autoLoad	A Boolean value that indicates whether the content loads automatically (true) or you must call Loader.load() (false).
Loader.bytesLoaded	A read-only property that indicates the number of bytes that have been loaded.
Loader.bytesTotal	A read-only property that indicates the total number of bytes in the content.
Loader.content	A reference to the content of the loader. This property is read-only.
Loader.contentPath	A string that indicates the URL of the content to be loaded.
Loader.percentLoaded	A number that indicates the percentage of loaded content. This property is read-only.
Loader.scaleContent	A Boolean value that indicates whether the content scales to fit the loader (true), or the loader scales to fit the content (false).

Event summary for the Screen class

The following table lists events of the Screen class.

Event	Description
Screen.allTransitionsInDone	Broadcast when all "in" transitions applied to a screen have finished.
Screen.allTransitionsOutDone	Broadcast when all "out" transitions applied to a screen have finished.
Screen.mouseDown	Broadcast when the mouse button was pressed over an object (shape or movie clip) directly owned by the screen.
Screen.mouseDownSomewhere	Broadcast when the mouse button was pressed somewhere on the Stage, but not necessarily on an object owned by this screen.
Screen.mouseMove	Broadcast when the mouse is moved while over a screen.
Screen.mouseOut	Broadcast when the mouse is moved from inside the screen to outside it.
Screen.mouseOver	Broadcast when the mouse is moved from outside this screen to inside it.
Screen.mouseUp	Broadcast when the mouse button was released over an object (shape or movie clip) directly owned by the screen.
Screen.mouseUpSomewhere	Broadcast when the mouse button was released somewhere on the Stage, but not necessarily over an object owned by this screen.

Events inherited from the UIObject class

The following table lists the events the Screen class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Screen class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.

Event	Description
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Events inherited from the Loader class

The following table lists the events the Screen class inherits from the Loader class.

Event	Description
Loader.complete	Triggered when the content finished loading.
Loader.progress	Triggered while content is loading.

Screen.allTransitionsInDone

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(allTransitionsInDone) {
  // your code here
}
listenerObject = new Object();
listenerObject.allTransitionsInDone = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("allTransitionsInDone", listenerObject)
```

Description

Event; broadcast when all "in" transitions applied to this screen have finished. The `allTransitionsInDone` event is broadcast by the Transition Manager associated with *screenObj*.

Example

In the following example, a button (`nextSlide_btn`) that's contained by the slide named `mySlide` is made visible when all the "in" transitions applied to `mySlide` have finished.

```
// Attached to mySlide:
on(allTransitionsInDone) {
  this.nextSlide_btn._visible = true;
}
```

See also

Screen.allTransitionsOutDone

Screen.allTransitionsOutDone

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(allTransitionsOutDone) {
  // your code here
}
listenerObject = new Object();
listenerObject.allTransitionsOutDone = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("allTransitionsOutDone", listenerObject)
```

Description

Event; broadcast when all "out" transitions applied to the screen have finished. The `allTransitionsOutDone` event is broadcast by the Transition Manager associated with *`screenObj`*.

See also

`Screen.currentFocusedScreen`

Screen.currentFocusedScreen

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myScreen.currentFocusedScreen
```

Description

Static property (read-only); returns a reference to the "leafmost" Screen object that contains the global current focus. *Leafmost* refers to the screen that is furthest away from the root screen in the screen hierarchy. The focus may be on the screen itself, or on a movie clip, text object, or component inside that screen. This property defaults to `null` if there is no current focus.

For example, assume you have a runtime screen hierarchy that looks like this:

```
presentation
  screen1
    subscreen1_1
      mymovieclip
        myUIButton
```

```
screen2
  subscreen1_2
```

If `myUIButton` has focus, the leafmost screen containing the focus is `subscreen1_1`, which is what `currentFocusedScreen` would return. In this case, `presentation`, `screen1`, and `subscreen1_1` all contain the focus but the one that is "closest" (in the screen hierarchy) to the leaves of the tree (that is, farthest away from the root) is `subscreen1_1`.

Example

The following example displays the name of the currently focused screen in the Output panel.

```
var currentFocus:mx.screens.Screen = mx.screens.Screen.currentFocusedScreen;
trace("Current screen is: " + currentFocus._name);
```

Screen.getChildScreen()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`myScreen``.getChildScreen(`*`childIndex`*`)`

Parameters

`childIndex` A number that indicates the zero-based index of the child screen to return.

Returns

A Screen object.

Description

Method; returns the child screen of *`myScreen`* whose index is *`childIndex`*.

Example

The following example sends the names of all the child screens belonging to the root screen named `Presentation` to the Output panel.

```
for (var i:Number = 0; i < _root.Presentation.numChildScreens; i++) {
    var childScreen:mx.screens.Screen = _root.Presentation.getChildScreen(i);
    trace(childScreen._name);
}
```

Screen.indexInParent

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myScreen.indexInParent
```

Description

Property (read-only); contains the zero-based index of *myScreen* in its parent's list of child screens.

Example

The following example displays the relative position of the screen `myScreen` in its parent screen's list of child screens.

```
var numChildren:Number = myScreen._parent.numChildScreens;
var myIndex:Number = myScreen.indexInParent;
trace("I'm child slide # " + myIndex + " out of " + numChildren + " screens.");
```

Screen.mouseDown

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(mouseDown) {
  // your code here
}
listenerObject = new Object();
listenerObject.mouseDown = function(eventObj){
  // insert your code here
}
screenObj.addEventListener("mouseDown", listenerObject)
```

Description

Event; broadcast when the mouse button is pressed over an object (for example, a shape or a movie clip) directly owned by the screen.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Example

The following code displays the name of the screen that captured the mouse event in the Output panel.

```
on(mouseDown) {
  trace("Mouse down event on: " + eventObj.target._name);
}
```

Screen.mouseDownSomewhere

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(mouseDownSomewhere) {
  // your code here
}
listenerObject = new Object();
listenerObject.mouseDownSomewhere = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("mouseDownSomewhere", listenerObject)
```

Description

Event; broadcast when the mouse button is pressed, but not necessarily over the specified screen.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Screen.mouseMove

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(mouseMove) {
  // your code here
}
listenerObject = new Object();
listenerObject.mouseMove = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("mouseMove", listenerObject)
```

Description

Event; broadcast when the mouse moves while over the screen. This event is sent only when the mouse is over the bounding box of this screen.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Note: This event may affect system performance and should be used judiciously.

Screen.mouseOut

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(mouseOut) {
  // your code here
}
listenerObject = new Object();
listenerObject.mouseOut = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("mouseOut", listenerObject)
```

Description

Event; broadcast when the mouse moves from inside the screen's bounding box to outside its bounding box.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Note: This event may affect system performance and should be used judiciously.

Screen.mouseOver

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(mouseOver) {
```

```
  // your code here
}
listenerObject = new Object();
listenerObject.mouseOver = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("mouseOver", listenerObject)
```

Description

Event; broadcast when the mouse moves from outside the screen's bounding box to inside its bounding box.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Note: This event may affect system performance and should be used judiciously.

Screen.mouseUp

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(mouseUp) {
  // your code here
}
listenerObject = new Object();
listenerObject.mouseUp = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("mouseUp", listenerObject)
```

Description

Event; broadcast when the mouse is released over the screen.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Screen.mouseUpSomewhere

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(mouseUpSomewhere) {
  // your code here
}
listenerObject = new Object();
listenerObject.mouseUpSomewhere = function(eventObject){
  // insert your code here
}
screenObj.addEventListener("mouseUpSomewhere", listenerObject)
```

Description

Event; broadcast when the mouse button is released, but not necessarily over the specified screen.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

Screen.numChildScreens

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myScreen.numChildScreens
```

Description

Property (read-only); returns the number of child screens contained by *myScreen*.

Example

The following example displays the names of all the child screens that belong to `myScreen`.

```
var howManyKids:Number = myScreen.numChildScreens;
for(i=0; i<howManyKids; i++) {
  var childScreen = myScreen.getChildScreen(i);
  trace(childScreen._name);
}
```

See also

`Screen.getChildScreen()`

Screen.parentIsScreen

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myScreen.parentIsScreen

Description

Property (read-only): returns a Boolean value indicating whether the specified screen's parent object is also a screen (`true`) or not (`false`). If this property is `false`, *myScreen* is at the root of its screen hierarchy.

Example

The following code determines if the parent object of the screen `myScreen` is also a screen. If `myScreen.parentIsScreen` is `true`, a `trace()` statement displays the number of sibling slides of `myScreen` in the Output panel. If the parent screen of `myScreen` is not also a screen, Flash assumes that `myScreen` is the root (master) slide in the presentation and therefore has no sibling slides.

```
if (myScreen.parentIsScreen) {
  trace("I have "+myScreen._parent.numChildScreens+" sibling screens");
} else {
  trace("I am the root screen and have no siblings");
}
```

Screen.parentScreen

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myScreen.parentScreen

Description

Property (read-only); returns the screen that contains *myScreen*. Returns `null` if *myScreen* is the root screen.

Example

The following example displays the name of the screen that contains the screen `myScreen`.

```
var myParent:mx.screens.Screen = myScreen.rootScreen;
```

Screen.rootScreen

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`myScreen``.rootScreen`

Description

Property (read-only); returns the screen at the top of the screen hierarchy that contains *`myScreen`*.

Example

The following example displays the name of the root screen that contains the screen `myScreen`.

```
var myRoot:mx.screens.Screen = myScreen.rootScreen;
```

ScrollPane component

The ScrollPane component displays movie clips, JPEG files, and SWF files in a scrollable area. By using a scroll pane, you can limit the amount of screen area occupied by these media types. The scroll pane can display content that is loaded from a local disk or from the Internet. You can set this content while authoring and at runtime by using ActionScript.

Once the scroll pane has focus, if its content has valid tab stops, those markers receive focus. After the last tab stop in the content, focus shifts to the next component. The vertical and horizontal scroll bars in the scroll pane never receive focus.

A ScrollPane instance receives focus if a user clicks it or tabs to it. When a ScrollPane instance has focus, you can use the following keys to control it:

Key	Description
Down Arrow	Content moves up one vertical line scroll.
End	Content moves to the bottom of the scroll pane.
Left Arrow	Content moves right one horizontal line scroll.
Home	Content moves to the top of the scroll pane.
Page Down	Content moves up one vertical page scroll.
Page Up	Content moves down one vertical page scroll.
Right Arrow	Content moves left one horizontal line scroll.
Up Arrow	Content moves down one vertical line scroll.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each ScrollPane instance reflects changes made to parameters in the Property inspector or Component inspector during authoring.

Using the ScrollPane component

You can use a scroll pane to display any content that is too large for the area into which it is loaded. For example, if you have a large image and only a small space for it in an application, you could load it into a scroll pane.

You can set up a scroll pane to allow users to drag the content within the pane by setting the `scrollDrag` parameter to `true`; a pointing hand appears on the content. Unlike most other components, events are broadcast when the mouse button is pressed and continue broadcasting until the button is released. If the contents of a scroll pane have valid tab stops, you must set `scrollDrag` to `false`; otherwise each mouse interaction with the contents will invoke scroll dragging.

ScrollPane parameters

You can set the following authoring parameters for each ScrollPane instance in the Property inspector or in the Component inspector:

contentPath indicates the content to load into the scroll pane. This value can be a relative path to a local SWF or JPEG file, or a relative or absolute path to a file on the Internet. It can also be the linkage identifier of a movie clip symbol in the library that is set to Export for ActionScript.

hLineScrollSize indicates the number of units a horizontal scroll bar moves each time an arrow button is clicked. The default value is 5.

hPageScrollSize indicates the number of units a horizontal scroll bar moves each time the track is clicked. The default value is 20.

hScrollPolicy displays the horizontal scroll bars. The value can be `on`, `off`, or `auto`. The default value is `auto`.

scrollDrag is a Boolean value that determines whether scrolling occurs (`true`) or not (`false`) when a user drags on the content within the scroll pane. The default value is `false`.

vLineScrollSize indicates the number of units a vertical scroll bar moves each time a scroll arrow is clicked. The default value is 5.

vPageScrollSize indicates the number of units a vertical scroll bar moves each time the scroll bar track is clicked. The default value is 20.

vScrollPolicy displays the vertical scroll bars. The value can be `on`, `off`, or `auto`. The default value is `auto`.

You can write ActionScript to control these and additional options for a ScrollPane component using its properties, methods, and events. For more information, see "ScrollPane class" on page 671.

Creating an application with the ScrollPane component

The following procedure explains how to add a ScrollPane component to an application while authoring. In this example, the scroll pane loads a SWF file that contains a logo.

To create an application with the ScrollPane component:

1. Drag a ScrollPane component from the Components panel to the Stage.
2. In the Property inspector, enter the instance name **myScrollPane**.
3. In the Property inspector, enter **logo.swf** for the contentPath parameter.
4. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
scrollListener = new Object();
scrollListener.scroll = function (evt){
  txtPosition.text = myScrollPane.vPosition;
}
myScrollPane.addEventListener("scroll", scrollListener);
completeListener = new Object;
completeListener.complete = function() {
  trace("logo.swf has completed loading.");
}
myScrollPane.addEventListener("complete", completeListener);
```

The first block of code is a `scroll` event handler on the `myScrollPane` instance that displays the value of the `vPosition` property in a TextField instance called `txtPosition`. The second block of code creates an event handler for the `complete` event that sends a message to the Output panel.

Customizing the ScrollPane component

You can transform a ScrollPane component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`) or any applicable properties and methods of the ScrollPane class.

Bear in mind these points about the ScrollPane component:

- The ScrollPane places the registration point of its content in the upper left corner of the pane.
- When the horizontal scroll bar is turned off, the vertical scroll bar is displayed from top to bottom along the right side of the scroll pane. When the vertical scroll bar is turned off, the horizontal scroll bar is displayed from left to right along the bottom of the scroll pane. You can also turn off both scroll bars.
- If the scroll pane is too small, the content may not display correctly.
- When the scroll pane is resized, the buttons remain the same size. The scroll track and scroll box (thumb) expand or contract, and their hit areas are resized.

Using styles with the ScrollPane component

The ScrollPane supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`border styles`	Both	The ScrollPane component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style is `"inset"`.
`scrollTrackColor`	Sample	The background color for the scroll track. The default value is 0xCCCCCC (light gray).
`symbolColor`	Sample	The color of the check mark. The default value is 0x000000 (black).
`symbolDisabledColor`	Sample	The color of the disabled check mark. The default value is 0x848384 (dark gray).

Using skins with the ScrollPane component

The ScrollPane component uses an instance of RectBorder for its border and scroll bars for scroll assets. For more information about skinning these visual elements, see "RectBorder class" on page 647 and "Using skins with the UIScrollBar component" on page 831.

ScrollPane class

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > ScrollPane

ActionScript Class Name mx.containers.ScrollPane

The properties of the ScrollPane class let you do the following at runtime: set the content, monitor the loading progress, and adjust the scroll amount.

Setting a property of the ScrollPane class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

You can set up a scroll pane so that users can drag the content within the pane. To do this, set the `scrollDrag` property to `true`; a pointing hand appears on the content. Unlike most other components, events are broadcast when the mouse button is pressed and continue broadcasting until the button is released. If the contents of a scroll pane have valid tab stops, you must set `scrollDrag` to `false`; otherwise, each mouse interaction with the contents will invoke scroll dragging.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.containers.ScrollPane.version);
```

Note: The code `trace(myScrollPaneInstance.version);` returns `undefined`.

Method summary for the ScrollPane class

The following table lists methods of the ScrollPane class.

Method	Description
`ScrollPane.getBytesLoaded()`	Returns the number of bytes of content loaded.
`ScrollPane.getBytesTotal()`	Returns the total number of bytes of content to be loaded.
`ScrollPane.refreshPane()`	Reloads the contents of the scroll pane.

Methods inherited from the UIObject class

The following table lists the methods the ScrollPane class inherits from the UIObject class. When calling these methods from the ScrollPane object, use the form *ScrollPaneInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the ScrollPane class inherits from the UIComponent class. When calling these methods from the ScrollPane object, use the form *ScrollPaneInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the ScrollPane class

The following table lists properties of the ScrollPane class.

Method	Description
ScrollPane.content	A reference to the content loaded into the scroll pane.
ScrollPane.contentPath	An absolute or relative URL of the SWF or JPEG file to load into the scroll pane.
ScrollPane.hLineScrollSize	The amount of content to scroll horizontally when a scroll arrow is clicked.
ScrollPane.hPageScrollSize	The amount of content to scroll horizontally when the scroll track is clicked.

Method	Description
ScrollPane.hPosition	The horizontal pixel position of the scroll pane's horizontal scroll bar.
ScrollPane.hScrollPolicy	The status of the horizontal scroll bar. It can be always on ("on"), always off ("off"), or on when needed ("auto"). The default value is "auto".
ScrollPane.scrollDrag	Indicates whether scrolling occurs (true) or not (false) when a user drags on content within the scroll pane. The default value is false.
ScrollPane.vLineScrollSize	The amount of content to scroll vertically when a scroll arrow is clicked.
ScrollPane.vPageScrollSize	The amount of content to scroll vertically when the scroll track is clicked.
ScrollPane.vPosition	The pixel position of the scroll pane's vertical scroll bar.
ScrollPane.vScrollPolicy	The status of the vertical scroll bar. It can be always on ("on"), always off ("off"), or on when needed ("auto"). The default value is "auto".

Properties inherited from the UIObject class

The following table lists the properties the ScrollPane class inherits from the UIObject class. When accessing these properties from the ScrollPane object, use the form *ScrollPaneInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the ScrollPane class inherits from the UIComponent class. When accessing these properties from the ScrollPane object, use the form *ScrollPaneInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the ScrollPane class

The following table lists events of the ScrollPane class.

Event	Description
ScrollPane.complete	Broadcast when the scroll pane content is loaded.
ScrollPane.progress	Broadcast while the scroll pane content is loading.
ScrollPane.scroll	Broadcast when the scroll bar is clicked.

Events inherited from the UIObject class

The following table lists the events the ScrollPane class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the ScrollPane class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

ScrollPane.complete

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(complete){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.complete = function(eventObject){
  ...
}
scrollPaneInstance.addEventListener("complete", listenerObject)
```

Description

Event; broadcast to all registered listeners when the content has finished loading.

The first usage example uses an `on()` handler and must be attached directly to a ScrollPane instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ScrollPane instance `myScrollPaneComponent`, sends "_level0.myScrollPaneComponent" to the Output panel:

```
on(complete){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`scrollPaneInstance`*) dispatches an event (in this case, `complete`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example creates a listener object with a `complete` event handler for the `scrollPane` instance:

```
form.complete = function(eventObj){
  // insert code to handle the event
}
scrollPane.addEventListener("complete",form);
```

ScrollPane.content

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`scrollPaneInstance``.content`

Description

Property (read-only); a reference to the content of the scroll pane. The value is `undefined` until the load begins.

Example

This example sets the `mcLoaded` variable to the value of the `content` property:

```
var mcLoaded = scrollPane.content;
```

See also

`ScrollPane.contentPath`

ScrollPane.contentPath

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`scrollPaneInstance``.contentPath`

Description

Property; a string that indicates an absolute or relative URL of the SWF or JPEG file to load into the scroll pane. A relative path must be relative to the SWF file that loads the content.

If you load content using a relative URL, the loaded content must be relative to the location of the SWF file that contains the scroll pane. For example, an application using a ScrollPane component that resides in the directory /scrollpane/nav/example.swf could load contents from the directory /scrollpane/content/flash/logo.swf by using the following `contentPath` property: `"../content/flash/logo.swf"`

Example

The following example tells the scroll pane to display the contents of an image from the Internet:

```
scrollPane.contentPath ="http://imagecache2.allposters.com/images/43/
  033_302.jpg";
```

The following example tells the scroll pane to display the contents of a symbol from the library:

```
scrollPane.contentPath ="movieClip_Name";
```

The following example tells the scroll pane to display the contents of the local file logo.swf:

```
scrollPane.contentPath ="logo.swf";
```

See also

`ScrollPane.content`

ScrollPane.getBytesLoaded()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`scrollPaneInstance``.getBytesLoaded()`

Parameters

None.

Returns

The number of bytes loaded in the scroll pane.

Description

Method; returns the number of bytes loaded in the ScrollPane instance. You can call this method at regular intervals while loading content to check its progress.

Example

This example creates a ScrollPane instance called `scrollPane`. It then defines a listener object called `loadListener` with a `progress` event handler that calls `getBytesLoaded()` to help determine the progress of the load:

```
createClassObject(mx.containers.ScrollPane, "scrollPane", 0);
loadListener = new Object();
```

```
loadListener.progress = function(eventObj){
  // eventObj.target is the component that generated the change event
  var bytesLoaded = scrollPane.getBytesLoaded();
  var bytesTotal = scrollPane.getBytesTotal();
  var percentComplete = Math.floor(bytesLoaded/bytesTotal);

  if (percentComplete < 5)  // loading begins
  {
    trace(" Starting loading contents from Internet");
  }
  else if(percentComplete = 50) // 50% complete
  {
    trace(" 50% contents downloaded ");
  }
}
scrollPane.addEventListener("progress", loadListener);
scrollPane.contentPath = "http://www.geocities.com/hcls_matrix/Images/
  homeview5.jpg";
```

ScrollPane.getBytesTotal()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollPaneInstance.getBytesTotal()

Parameters

None.

Returns

A number.

Description

Method; returns the total number of bytes to be loaded into the ScrollPane instance.

See also

ScrollPane.getBytesLoaded()

ScrollPane.hLineScrollSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.hLineScrollSize
```

Description

Property; the number of pixels to move the content when an arrow in the horizontal scroll bar is clicked. The default value is 5.

Example

This example increases the horizontal scroll unit to 10:

```
scrollPane.hLineScrollSize = 10;
```

ScrollPane.hPageScrollSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.hPageScrollSize
```

Description

Property; the number of pixels to move the content when the track in the horizontal scroll bar is clicked. The default value is 20.

Example

This example increases the horizontal page scroll unit to 30:

```
scrollPane.hPageScrollSize = 30;
```

ScrollPane.hPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.hPosition
```

Description

Property; the pixel position of the scroll pane's horizontal scroll box (thumb). The 0 position is at the extreme left end of the scroll track, which causes the left edge of the scroll pane content to be visible in the scroll pane.

Example

This example positions the scroll bar at pixel 20:

```
scrollPane.hPosition = 20;
```

ScrollPane.hScrollPolicy

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollPaneInstance.hScrollPolicy

Description

Property; determines whether the horizontal scroll bar is always present (`"on"`), is never present (`"off"`), or appears automatically according to the size of the image (`"auto"`). The default value is `"auto"`.

Example

The following code turns scroll bars on all the time:

```
scrollPane.hScrollPolicy = "on";
```

ScrollPane.progress

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(progress){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.progress = function(eventObject){
  ...
}
scrollPaneInstance.addEventListener("progress", listenerObject)
```

Description

Event; broadcast to all registered listeners while content is loading. The progress event is not always broadcast; the `complete` event may be broadcast without any `progress` events being dispatched. This can happen especially if the loaded content is a local file. Your application triggers the `progress` event when the content starts loading by setting the value of the `contentPath` property.

The first usage example uses an `on()` handler and must be attached directly to a ScrollPane instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the ScrollPane component instance `mySPComponent`, sends "_level0.mySPComponent" to the Output panel:

```
on(progress){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`scrollPaneInstance`*) dispatches an event (in this case, `progress`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following code creates a ScrollPane instance called `scrollPane`. It then creates a listener object with an event handler for the `progress` event that sends a message to the Output panel about how much content has loaded.

```
createClassObject(mx.containers.ScrollPane, "scrollPane", 0);
loadListener = new Object();
loadListener.progress = function(eventObj){
  // eventObj.target is the component that generated the progress event
  // --in this case, scrollPane
  trace("logo.swf has loaded " + scrollPane.getBytesLoaded() + " Bytes.");
  // track loading progress
}
scrollPane.addEventListener("complete", loadListener);
scrollPane.contentPath = "logo.swf";
```

ScrollPane.refreshPane()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.refreshPane()
```

Parameters

None.

Returns

Nothing.

Description

Method; refreshes the scroll pane after content is loaded. This method reloads the content. You could use this method if, for example, you've loaded a form into a scroll pane and an input property (for example, a text field) has been changed by ActionScript. In this case, you would call `refreshPane()` to reload the same form with the new values for the input properties.

Example

The following example refreshes the scroll pane instance `sp`:

```
sp.refreshPane();
```

ScrollPane.scroll

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(scroll){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.scroll = function(eventObject){
  ...
}
scrollPaneInstance.addEventListener("scroll", listenerObject)
```

Event object

In addition to the standard event object properties, there are two additional properties defined for the `scroll` event: a `type` property whose value is `"scroll"`, and a `direction` property whose value can be `"vertical"` or `"horizontal"`.

In addition to the standard event object properties, there are two additional properties defined for the `ProgressBar.progress` event: `current` (the loaded value equals `total`), and `total` (the total value).

Description

Event; broadcast to all registered listeners when a user clicks the scroll bar buttons, scroll box (thumb), or scroll track. Unlike other events, the `scroll` event is broadcast when a user presses the mouse button on the scroll bar and continues broadcasting until the mouse is released.

The first usage example uses an `on()` handler and must be attached directly to a ScrollPane instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the instance `sp`, sends "_level0.sp" to the Output panel:

```
on(scroll){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`scrollPaneInstance`*) dispatches an event (in this case, `scroll`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example creates a `form` listener object with a `scroll` callback function that's registered to the `spInstance` instance. You must fill `spInstance` with content.

```
spInstance.contentPath = "mouse3.jpg";
form = new Object();
form.scroll = function(eventObj){
  trace("ScrollPane scrolled");
}
spInstance.addEventListener("scroll", form);
```

See also

`EventDispatcher.addEventListener()`

ScrollPane.scrollDrag

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollPaneInstance.scrollDrag

Description

Property; a Boolean value that indicates whether scrolling occurs (`true`) or not (`false`) when a user drags within the scroll pane. The default value is `false`.

Example

This example causes the content to scroll when the user drags within the scroll pane:

```
scrollPane.scrollDrag = true;
```

ScrollPane.vLineScrollSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollPaneInstance.vLineScrollSize

Description

Property; the number of pixels to move the content in the display area when the user clicks a scroll arrow in a vertical scroll bar. The default value is 5.

Example

This code causes the content in the display area to move 10 pixels when the vertical scroll arrows are clicked:

```
scrollPane.vLineScrollSize = 10;
```

ScrollPane.vPageScrollSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.vPageScrollSize
```

Description

Property; the number of pixels to move the content in the display area when the user clicks the track in a vertical scroll bar. The default value is 20.

Example

This code causes the content in the display area to move 30 pixels when the vertical scroll track is clicked:

```
scrollPane.vPageScrollSize = 30;
```

ScrollPane.vPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.vPosition
```

Description

Property; the pixel position of the scroll pane's vertical scroll bar. The default value is 0.

ScrollPane.vScrollPolicy

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.vScrollPolicy
```

Description

Property; determines whether the vertical scroll bar is always present (`"on"`), is never present (`"off"`), or appears automatically according to the size of the image (`"auto"`). The default value is `"auto"`.

Example

The following code turns vertical scroll bars on all the time:

```
scrollPane.vScrollPolicy = "on";
```

SimpleButton class

Inheritance MovieClip > UIObject class > UIComponent class > SimpleButton

ActionScript Class Name mx.controls.SimpleButton

The properties of the SimpleButton class let you control the following at runtime:

- Whether a button has the emphasized look of a default push button
- Whether the button acts as a push button or as a toggle switch
- Whether a button is selected

Method summary for the SimpleButton class

There are no methods exclusive to the SimpleButton class.

Methods inherited from the UIObject class

The following table lists the methods the SimpleButton class inherits from the UIObject class. When calling these methods from the SimpleButton class, use the form *buttonInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the SimpleButton class inherits from the UIComponent class. When calling these methods from the SimpleButton object, use the form *buttonInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the SimpleButton class

The following table lists properties of the SimpleButton class.

Property	Description
SimpleButton.emphasized	Indicates whether a button has the appearance of a default push button.
SimpleButton.emphasizedStyleDeclaration	The style declaration when the `emphasized` property is set to `true`.
SimpleButton.selected	A Boolean value indicating whether the button is selected (`true`) or not (`false`). The default value is `false`.
SimpleButton.toggle	A Boolean value indicating whether the button behaves as a toggle switch (`true`) or not (`false`). The default value is `false`.

Properties inherited from the UIObject class

The following table lists the properties the SimpleButton class inherits from the UIObject class. When accessing these properties from the SimpleButton object, use the form *`buttonInstance.propertyName`*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the SimpleButton class inherits from the UIComponent class. When accessing these properties from the SimpleButton object, use the form *buttonInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the SimpleButton class

The following table lists the event of the SimpleButton class.

Event	Description
SimpleButton.click	Broadcast when a button is clicked.

Events inherited from the UIObject class

The following table lists the events the SimpleButton class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the SimpleButton class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

SimpleButton.click

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(click){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.click = function(eventObject){
  ...
}
buttonInstance.addEventListener("click", listenerObject)
```

Description

Event; broadcast to all registered listeners when the mouse is clicked (released) over the button or if the button has focus and the Spacebar is pressed.

The first usage example uses an `on()` handler and must be attached directly to a Button component instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the Button component instance `myButtonComponent`, sends "_level0.myButtonComponent" to the Output panel:

```
on(click){
  trace(this);
}
```

The behavior of `this` is different when used inside an `on()` handler attached to a regular Flash button symbol. In that situation, `this` refers to the Timeline that contains the button. For example, the following code, attached to the button symbol instance `myButton`, sends "_level0" to the Output panel:

```
on(release){
  trace(this);
}
```

Note: The built-in ActionScript Button object doesn't have a `click` event; the closest event is `release`.

The second usage example uses a dispatcher/listener event model. A component instance (*buttonInstance*) dispatches an event (in this case, `click`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event occurs. When the event occurs, it automatically passes an event object (*eventObject*) to the listener object method. The event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call `addEventListener()` (see `EventDispatcher.addEventListener()`) on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a button called `buttonInstance` is clicked. The first line specifies that the button act like a toggle switch. The second line creates a listener object called `form`. The third line defines a function for the `click` event on the listener object. Inside the function is a `trace()` statement that uses the event object that is automatically passed to the function (in this example, `eventObj`) to generate a message. The `target` property of an event object is the component that generated the event (in this example, `buttonInstance`). The `SimpleButton.selected` property is accessed from the event object's `target` property. The last line calls `addEventListener()` from `buttonInstance` and passes it the `click` event and the `form` listener object as parameters.

```
buttonInstance.toggle = true;
form = new Object();
form.click = function(eventObj){
  trace("The selected property has changed to " + eventObj.target.selected);
}
buttonInstance.addEventListener("click", form);
```

The following code also sends a message to the Output panel when `buttonInstance` is clicked. The `on()` handler must be attached directly to `buttonInstance`.

```
on(click){
  trace("button component was clicked");
}
```

SimpleButton.emphasized

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

buttonInstance.emphasized

Description

Property; indicates whether the button is in an emphasized state (`true`) or not (`false`). The emphasized state is equivalent to the appearance of a default push button. In general, use the `FocusManager.defaultPushButton` property instead of setting the `emphasized` property directly. The default value is `false`.

If you aren't using `FocusManager.defaultPushButton`, you might just want to set a button to the emphasized state, or use the emphasized state to change text from one color to another. The following example sets the `emphasized` property for the button instance `myButton`:

```
_global.styles.foo = new CSSStyleDeclaration();
_global.styles.foo.color = 0xFF0000;
SimpleButton.emphasizedStyleDeclaration = "neutralStyle";
myButton.emphasized = true;
```

See also

`SimpleButton.emphasizedStyleDeclaration`

SimpleButton.emphasizedStyleDeclaration

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`buttonInstance``.emphasizedStyleDeclaraion`

Description

Property (static); a string indicating the style declaration that formats a button when the `emphasized` property is set to `true`.

The `emphasizedStyleDeclaration` property is a static property of the SimpleButton class. Therefore, you must access it directly from SimpleButton, rather than from a *`buttonInstance`*, as in the following:

```
SimpleButton.emphasizedStyleDeclaration = "3dEmphStyle";
```

See also

`SimpleButton.emphasized`

SimpleButton.selected

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
buttonInstance.selected
```

Description

Property; a Boolean value that indicates whether the button is selected (`true`) or not (`false`). The default value is `false`.

SimpleButton.toggle

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
buttonInstance.toggle
```

Description

Property; a Boolean value that indicates whether the button acts as a toggle switch (`true`) or not (`false`). The default value is `false`.

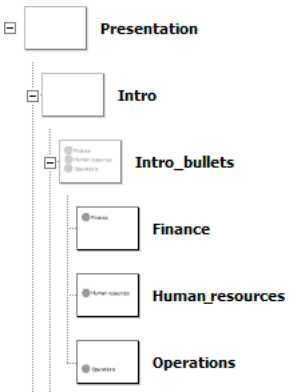
If a button acts as a toggle switch, it stays pressed until you click it again to release it.

Slide class (Flash Professional only)

The Slide class corresponds to a node in a hierarchical slide presentation. In Flash MX Professional 2004, you can create slide presentations using the Screen Outline pane. For an overview of working with screens, see Chapter 12, "Working with Screens (Flash Professional Only)," in *Using Flash*.

The Slide class extends the Screen class (see "Screen class (Flash Professional only)" on page 651), and provides built-in navigation and sequencing capabilities between slides, as well as the ability to easily attach transitions between slides using behaviors. Slides maintain "state," so that when a user advances to an adjacent slide, the previous slide is hidden.

Note that you can only navigate to ("stop on") slides that don't contain any child slides ("leaf" slides). For example, consider the following illustration, which shows the contents of the Screen Outline pane for a sample slide presentation:



When this presentation starts, it will, by default, "stop" on the slide named Finance, which is the first slide in the presentation that doesn't contain any child slides.

Also note that child slides "inherit" the visual appearance (graphics and other content) of their parent slides. For example, in the above illustration, in addition to the content on the Finance slide, the user would also see any content on the Intro and Presentation slides.

Note: The Slide class inherits from the Loader class, which lets you easily load external SWF or JPEG files into a given slide. This provides a way to make your slide presentations modular and reduce initial download time. For more information, see "Loading external content into screens (Flash Professional only)" on page 651.

Using the Slide class (Flash Professional only)

You use the methods and properties of the Slide class to control slide presentations you create using the Screen Outline pane (Window > Screen), to get information about a slide presentation (for example, to determine the number of child slides contained by parent slide), or to navigate between slides in a slide presentation (for example, to create "Next slide" and "Previous slide" buttons).

You can also use the built-in behaviors that are available in the Behaviors panel to control slide presentations. For more information, see "Adding controls to screens using behaviors (Flash Professional only)" in *Using Flash*.

Slide parameters

You can set the following authoring parameters for each slide in the Property inspector or in the Component inspector:

autoKeyNav determines how, or if, the slide responds to the default keyboard navigation. For more information, see `Slide.autoKeyNav`.

autoload indicates whether the content specified by the contentPath parameter should load automatically (true), or wait to load until the `Loader.load()` method is called (false). The default value is true.

contentPath specifies the contents of the slide. This can be the linkage identifier of a movie clip or an absolute or relative URL of a SWF or JPEG file to load into the slide. By default, loaded content is clipped to fit the slide.

overlayChildren specifies whether the slide's child slides remain visible (true) or not (false) when you navigate from one child slide to the next.

playHidden specifies whether the slide continues to play (true) or not (false) when hidden.

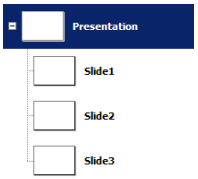
Using the Slide class to create a slide presentation

You use the methods and properties of the Slide class to control slide presentations you create in the Screen Outline pane (Window > Screen) in the Flash authoring environment. (The Behaviors panel also contains several behaviors for creating slide navigation.) In this example, you write your own ActionScript to create Next and Previous buttons for a slide presentation.

To create a slide presentation with navigation:

1. In Flash, select File > New.
2. On the General tab, select Flash Slide Presentation.
3. In the Screen Outline pane, click the Insert Screen (+) button twice to create two new slides beneath the Presentation slide.

   The Screen Outline pane should look like the following:



4. Select Slide1 in the Screen Outline pane and, using the Text tool, add a text field that reads **This is slide one**.
5. Repeat the previous step for Slide2 and Slide3, creating text fields on each slide that read **This is slide two** and **This is slide three**, respectively.

6. Select the Presentation slide and open the Components panel.
7. Drag a Button component from the Components panel to the bottom of the Stage.
8. In the Property inspector, type **Next Slide** for the Button component's Label property.
9. In the Actions panel, type the following code:

```
on(click) {
  _parent.currentSlide.gotoNextSlide();
}
```

10. Test the SWF file (Control > Test Movie) and click the Next Slide button to advance to the next slide.

Slide class (API) (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > Loader component > Screen class (Flash Professional only) > Slide

ActionScript Class Name mx.screens.Slide

The methods, properties, and events of the Slide class allow you to manage and manipulate slides.

Method summary for the Slide class

The following table lists methods of the Slide class:

Method	Description
`Slide.getChildSlide()`	Returns the specified child slide.
`Slide.gotoFirstSlide()`	Navigates to the first leaf slide in the slide's hierarchy of subslides.
`Slide.gotoLastSlide()`	Navigates to the last leaf slide in the slide's hierarchy of subslides.
`Slide.gotoNextSlide()`	Navigates to the next slide.
`Slide.gotoPreviousSlide()`	Navigates to the previous slide.
`Slide.gotoSlide()`	Navigates to an specified slide.

Methods inherited from the UIObject class

The following table lists the methods the Slide class inherits from the UIObject class. When calling these methods from the Slide object, use the form *`SlideInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.

Method	Description
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Slide class inherits from the UIComponent class. When calling these methods from the Slide object, use the form *SlideInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Methods inherited from the Loader class

The following table lists the method the Slide class inherits from the Loader class. When calling this method from the Slide object, use the form *SlideInstance.methodName*.

Method	Description
Loader.load()	Loads the content specified by the contentPath property.

Methods inherited from the Screen class

The following table lists the method the Slide class inherits from the Screen class. When calling this method from the Slide object, use the form *SlideInstance.methodName*.

Method	Description
Screen.getChildScreen()	Returns the child screen of this screen at a particular index.

Property summary for the Slide class

The following table lists properties of the Slide class:

Property	Description
Slide.autoKeyNav	Determines whether the slide uses default keyboard handling to navigate to the next/previous slide.
Slide.currentChildSlide	Read-only; returns the immediate child of the specified slide that contains the currently active slide.
Slide.currentFocusedSlide	Read-only; returns the "leafmost" slide (the slide farthest from the root of the slide tree) that contains the global current focus.

Property	Description
Slide.currentSlide	Read-only; returns the currently active slide.
Slide.defaultKeydownHandler	Callback handler that overrides the default keyboard navigation (Left and Right Arrow keys).
Slide.firstSlide	Read-only; returns the slide's first child slide that has no children.
Slide.indexInParentSlide	Read-only; returns the slide's index (zero-based) in its parent's list of subslides.
Slide.lastSlide	Read-only; returns the slide's last child slide that has no children.
Slide.nextSlide	Read-only; returns the slide you would reach if you called `mySlide.gotoNextSlide()`, but does not actually navigate to that slide.
Slide.numChildSlides	Read-only; returns the number of child slides the slide contains.
Slide.overlayChildren	Determines whether the slide's child slides are visible when control flows from one child slide to the next.
Slide.parentIsSlide	Read-only; returns a Boolean value indicating whether the parent object of the slide is also a slide (`true`) or not (`false`).
Slide.playHidden	Determines whether the slide continues to play when hidden.
Slide.previousSlide	Read-only; returns the slide you would reach if you called `mySlide.gotoPreviousSlide()`, but does not actually navigate to that slide.
Slide.rootSlide	Read-only; returns the root of the slide tree that contains the slide.

Properties inherited from the UIObject class

The following table lists the properties the Slide class inherits from the UIObject class. When accessing these properties from the Slide object, use the form *`SlideInstance.propertyName`*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.

Property	Description
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Slide class inherits from the UIComponent class. When accessing these properties from the Slide object, use the form *SlideInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the Loader class

The following table lists the properties the Slide class inherits from the Loader class. When accessing these properties from the Slide object, use the form *SlideInstance.propertyName*.

Property	Description
Loader.autoLoad	A Boolean value that indicates whether the content loads automatically (true) or you must call Loader.load() (false).
Loader.bytesLoaded	A read-only property that indicates the number of bytes that have been loaded.
Loader.bytesTotal	A read-only property that indicates the total number of bytes in the content.
Loader.content	A reference to the content of the loader. This property is read-only.
Loader.contentPath	A string that indicates the URL of the content to be loaded.
Loader.percentLoaded	A number that indicates the percentage of loaded content. This property is read-only.
Loader.scaleContent	A Boolean value that indicates whether the content scales to fit the loader (true), or the loader scales to fit the content (false).

Properties inherited from the Screen class

The following table lists the properties the Slide class inherits from the Screen class. When accessing these properties from the Slide object, use the form *SlideInstance.propertyName*.

Property	Description
Screen.currentFocusedScreen	Read-only; returns the screen that contains the global current focus.
Screen.indexInParent	Read-only; returns the screen's index (zero-based) in its parent screen's list of child screens.
Screen.numChildScreens	Read-only; returns the number of child screens contained by the screen.
Screen.parentIsScreen	Read-only; returns a Boolean (`true` or `false`) value that indicates whether the screen's parent object is itself a screen.
Screen.rootScreen	Read-only; returns the root screen of the tree or subtree that contains the screen.

Event summary for the Slide class

The following table lists events of the Slide class.

Event	Description
Slide.hideChild	Broadcast each time a child of a slide changes from visible to invisible.
Slide.revealChild	Broadcast each time a child slide of a slide object changes from invisible to visible.

Events inherited from the UIObject class

The following table lists the events the Slide class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Slide class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

Events inherited from the Loader class

The following table lists the events the Slide class inherits from the Loader class.

Event	Description
`Loader.complete`	Triggered when the content finished loading.
`Loader.progress`	Triggered while content is loading.

Events inherited from the Screen class

The following table lists the events the Slide class inherits from the Screen class.

Event	Description
`Screen.allTransitionsInDone`	Broadcast when all “in” transitions applied to a screen have finished.
`Screen.allTransitionsOutDone`	Broadcast when all “out” transitions applied to a screen have finished.
`Screen.mouseDown`	Broadcast when the mouse button was pressed over an object (shape or movie clip) directly owned by the screen.
`Screen.mouseDownSomewhere`	Broadcast when the mouse button was pressed somewhere on the Stage, but not necessarily on an object owned by this screen.
`Screen.mouseMove`	Broadcast when the mouse is moved while over a screen.
`Screen.mouseOut`	Broadcast when the mouse is moved from inside the screen to outside it.
`Screen.mouseOver`	Broadcast when the mouse is moved from outside this screen to inside it.
`Screen.mouseUp`	Broadcast when the mouse button was released over an object (shape or movie clip) directly owned by the screen.
`Screen.mouseUpSomewhere`	Broadcast when the mouse button was released somewhere on the Stage, but not necessarily over an object owned by this screen.

Slide.autoKeyNav

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.autoKeyNav

Description

Property; determines whether the slide uses default keyboard handling to navigate to the next/previous slide when *mySlide* has focus. This property accepts the string values "true", "false", and "inherit". You can override this default keyboard handling behavior by using the Slide.defaultKeydownHandler property.

When the value of this property is "true", pressing the Right Arrow key (Key.RIGHT) or the Spacebar (Key.SPACE) when *mySlide* has focus advances to the next slide; pressing the Left Arrow key (Key.Left) moves to the previous slide.

When this property is set to "false", no default keyboard handling takes place when *mySlide* has focus.

When this property is set to "inherit", *mySlide* checks the autoKeyNav property of its parent slide. If it is also set to "inherit", Flash looks up the slide inheritance chain until it finds a parent slide whose autoKeyNav property is set to "true" or "false".

If *mySlide* has no parent slide (that is, if the statement (mySlide.parentIsSlide == false) is true), it behaves as if autoKeyNav had been set to "true".

Example

This example turns off automatic keyboard navigation for the slide named loginSlide.

```
_root.Presentation.loginSlide.autoKeyNav = "false";
```

See also

Slide.defaultKeydownHandler

Slide.currentChildSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.currentChildSlide

Description

Property (read-only); returns the immediate child of *mySlide* that contains the currently active slide; returns `null` if no child slide contained by *mySlide* has the current focus.

Example

Consider the following screen outline:

```
Presentation
  Slide_1
    Bullet1_1
      SubBullet1_1_1
    Bullet1_2
      SubBullet1_2_1
  Slide_2
```

Assuming that `SubBullet1_1_1` is the current slide, then the following statements are all true:

```
Presentation.currentChildSlide == Slide_1;
Slide_1.currentChildSlide == Bullet_1_1;
SubBullet_1_1_1.currentChildSlide == null;
Slide_2.currentChildSlide == null;
```

See also

`Slide.currentSlide`

Slide.currentFocusedSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
mx.screens.Slide.currentFocusedSlide
```

Description

Property (read-only); returns the "leafmost" slide (the slide farthest from the root of the slide tree) that contains the current global focus. The actual focus may be on the slide itself, or on a movie clip, text object, or component inside that slide; the method returns `null` if there is no current focus.

Example

```
var focusedSlide = mx.screens.Slide.currentFocusedSlide;
```

Slide.currentSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
mySlide.currentSlide
```

Description

Property (read-only); returns the currently active slide. This is always a "leaf" slide—that is, a slide that contains no child slides.

Example

The following code, attached to a button on the root presentation slide, advances the slide presentation to the next slide each time the button is clicked.

```
// Attached to button instance contained by presentation slide:
on(press) {
  _parent.currentSlide.gotoNextSlide();
}
```

See also

`Slide.gotoNextSlide()`

Slide.defaultKeydownHandler

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
mySlide.defaultKeyDownHandler = function (eventObj) {
  // your code here
}
```

Parameters

eventObj An event object with the following properties:

- `type` A string indicating the type of event. Possible values are `"keyUp"` and `"keyDown"`.
- `ascii` An integer that represents the ASCII value of the last key pressed; corresponds to the value returned by `Key.getAscii()`.
- `code` An integer that represents the key code of the last key pressed; corresponds to the value returned by `Key.getCode()`.

- `shiftKey` A Boolean value indicating if the Shift key is currently being pressed (`true`) or not (`false`).
- `ctrlKey` A Boolean value indicating if the Control key is currently being pressed (`true`) or not (`false`).

Returns

Nothing.

Description

Callback handler; lets you override the default keyboard navigation with a custom keyboard handler that you create. For example, instead of having the Left and Right Arrow keys navigate to the previous and next slides in a presentation, respectively, you could have the Up and Down Arrow keys perform those functions. For a discussion of the default keyboard handling behavior, see `Slide.autoKeyNav`.

Example

In that example, the default keyboard handling is altered for child slides of the slide to which the `on(load)` handler is attached. This handler uses the Up and Down Arrow keys for navigation instead of the Left and Right Arrow keys.

```
on (load) {
  this.defaultKeyDownHandler = function(eventObj:Object) {
    switch (eventObj.code) {
    case Key.DOWN :
      this.currentSlide.gotoNextSlide();
      break;
    case Key.UP :
      this.currentSlide.gotoPreviousSlide();
      break;
    default :
      break;
    }
  };
}
```

See also

`Slide.autoKeyNav`

Slide.firstSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.firstSlide

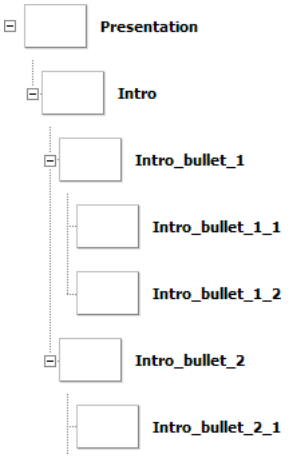
Description

Property (read-only); returns the first child slide of *mySlide* that has no child slides.

Example

In the hierarchy of slides shown below, the following statements are both true:

```
Presentation.Intro.firstSlide == Intro_bullet_1_1;
Presentation.Intro_bullet_1.firstSlide == Intro_bullet_1-1;
```



Slide.getChildSlide()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.getChildSlide(*childIndex*)

Parameters

childIndex The zero-based index of the child slide to return.

Returns

A slide object.

Description

Method; returns the child slide of *mySlide* whose index is *childIndex*. You can use this method to iterate over a set of child slides whose indices are known.

Example

The following code causes the Output panel to display the names of all the child slides of the root presentation slide.

```
var numSlides = _root.Presentation.numChildSlides;
for(var slideIndex=0; slideIndex < numSlides; slideIndex++) {
  var childSlide = _root.Presentation.getChildSlide(slideIndex);
  trace(childSlide._name);
}
```

See also

`Slide.numChildSlides`

Slide.gotoFirstSlide()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoFirstSlide()`

Parameters

None.

Returns

Nothing.

Description

Method; navigates to the first leaf slide in the tree of child slides beneath *`mySlide`*. This method is ignored when called from within a slide's `on(hide)` or `on(reveal)` event handler if that event was a result of a slide navigation.

To go to the first slide in a presentation, call *`mySlide`*`.rootSlide.gotoFirstSlide()`. (For more information on `rootSlide`, see `Slide.revealChild`.)

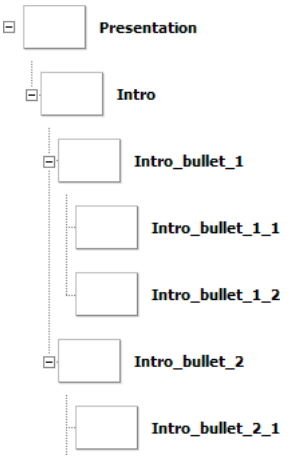
Example

In the slide hierarchy illustrated below, the following method calls would all navigate to the slide named `Intro_bullet_1_1`:

```
Presentation.gotoFirstSlide();
Presentation.Intro.gotoFirstSlide();
Presentation.Intro.Intro_bullet_1.gotoFirstSlide();
```

This method call would navigate to the slide named `Intro_bullet_2_1`:

```
Presentation.Intro.Intro_bullet_2.gotoFirstSlide();
```



See also

`Slide.firstSlide`, `Slide.revealChild`

Slide.gotoLastSlide()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoLastSlide()`

Parameters

None.

Returns

Nothing.

Description

Method; navigates to the last leaf slide in the tree of child slides beneath *`mySlide`*. This method is ignored when called from within a slide's `on(hide)` or `on(reveal)` event handler if that event was a result of another slide navigation.

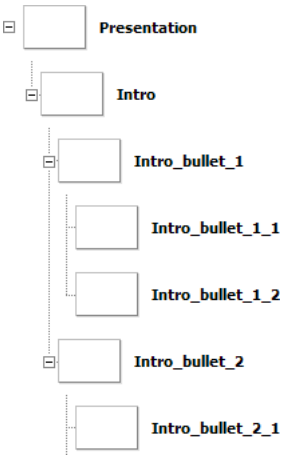
Example

In the slide hierarchy illustrated below, the following method calls would navigate to the slide named `Intro_bullet_1_2`:

```
Presentation.Intro.gotoLastSlide();
Presentation.Intro.Intro_bullet_1.gotoLastSlide();
```

These method calls would navigate to the slide named `Intro_bullet_2_1`:

```
Presentation.gotoLastSlide();
Presentation.Intro.gotoLastSlide();
```



See also

`Slide.gotoSlide()`, `Slide.lastSlide`

Slide.gotoNextSlide()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoNextSlide()`

Parameters

None.

Returns

A Boolean value, or `null`. The method returns `true` if it successfully navigated to the next slide; it returns `false` if the presentation is already at the last slide when the method is invoked (that is, if `currentSlide.nextSlide` is `null`). The method returns `null` if invoked on a slide that doesn't contain the current slide.

Description

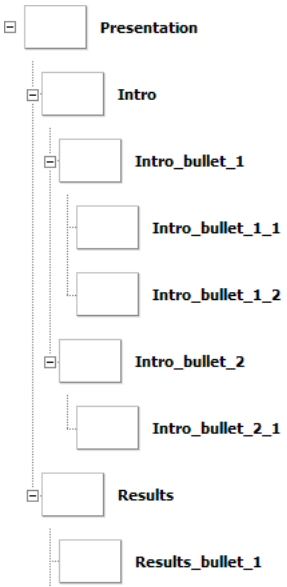
Method; navigates to the next slide in the slide presentation. As control passes from one slide to the next, the outgoing slide is hidden and the incoming slide is revealed. If the outgoing and incoming slides are in different slide subtrees, then all ancestor slides, starting with the outgoing slide and up to the common ancestor of the incoming and outgoing slides, are hidden and receive a `hide` event. Immediately following, all ancestor slides of the incoming slide, up to the common ancestor of the outgoing and incoming slide, are made visible and receive a `reveal` event.

Typically, `gotoNextSlide()` is called on the leaf node that represents the current slide. If called on a nonleaf node, `someNode`, then `someNode.gotoNextSlide()` advances to the first leaf node in the next slide or "section."

This method has no effect when invoked on a slide that does not contain the current slide. This method also has no effect when called from within an `on(hide)` or `on(reveal)` event handler attached to a slide, if that handler was invoked as a result of slide navigation.

Example

Suppose that, in the following slide hierarchy, the slide named `Intro_bullet_1_1` is the current slide being viewed (that is, `_root.Presentation.currentSlide._name == Intro_bullet_1_1`).



In this case, calling `Intro_bullet_1_1.gotoNextSlide()` would navigate to `Intro_bullet_1_2`, which is a sibling slide of `Intro_bullet_1_1`.

However, calling `Intro_bullet_1.gotoNextSlide()` would navigate to `Intro_bullet_2_1`, the first leaf slide contained by `Intro_bullet_2`, which is the next sibling slide of `Intro_bullet_1`. Similarly, calling `Intro.gotoNextSlide()` would navigate to `Results_bullet_1`, the first leaf slide contained by the `Results` slide.

Also, still assuming that the current slide is `Intro_bullet_1_1`, calling `Results.gotoNextSlide()` would have no effect, because `Results` does not contain the current slide (that is, `Results.currentSlide` is `null`).

See also

`Slide.currentSlide`, `Slide.gotoPreviousSlide()`, `Slide.nextSlide`

Slide.gotoPreviousSlide()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoPreviousSlide()`

Parameters

None.

Returns

A Boolean value, or `null`. The method returns `true` if it successfully navigated to the previous slide; it returns `false` if the presentation is at the first slide when the method is invoked (that is, if `currentSlide.nextSlide` is `null`). The method returns `null` if invoked on a slide that doesn't contain the current slide.

Description

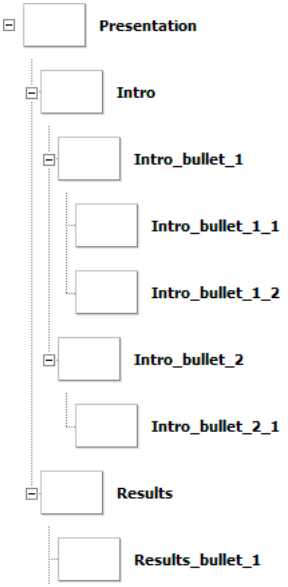
Method; navigates to the previous slide in the slide presentation. As control passes from one slide to the previous slide, the outgoing slide is hidden and the incoming slide is revealed. If the outgoing and incoming slides are in different slide subtrees, then all ancestor slides, starting with the outgoing slide and up to the common ancestor of the incoming and outgoing slides, are hidden and receive a `hide` event. Immediately following, all ancestors slides of the incoming slide, up to the common ancestor of the outgoing and incoming slide, are made visible and receive a `reveal` event.

Typically, `gotoPreviousSlide()` is called on the leaf node that represents the current slide. If called on a nonleaf node, `someNode`, then `someNode.gotoPreviousSlide()` advances to the first leaf node in the previous slide or "section."

This method has no effect when invoked on a slide that does not contain the current slide. This method also has no effect when called from within an `on(hide)` or `on(reveal)` event handler attached to a slide, if that handler was invoked as a result of slide navigation.

Example

Suppose that, in the following slide hierarchy, the slide named `Intro_bullet_1_2` is the current slide being viewed (that is, `_root.Presentation.currentSlide._name == Intro_bullet_1_2`).



In this case, calling `Intro_bullet_1_2.gotoPreviousSlide()` would navigate to `Intro_bullet_1_1`, which is the previous sibling slide of `Intro_bullet_1_2`.

However, calling `Intro_bullet_2.gotoPreviousSlide()` would navigate to `Intro_bullet_1_1`, the first leaf slide contained by `Intro_bullet_1`, which is the previous sibling slide of `Intro_bullet_2`. Similarly, calling `Results.gotoPreviousSlide()` would navigate to `Intro_bullet_1_1`, the first leaf slide contained by the `Intro` slide.

Also, if the current slide is `Intro_bullet_1_1`, then calling `Results.gotoPreviousSlide()` would have no effect, since `Results` does not contain the current slide (that is, `Results.currentSlide` is `null`).

See also

`Slide.currentSlide`, `Slide.gotoNextSlide()`, `Slide.previousSlide`

Slide.gotoSlide()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
mySlide.gotoSlide(newSlide)
```

Parameters

newSlide The slide to navigate to.

Returns

A Boolean value indicating if the navigation succeeded (`true`) or not (`false`).

Description

Method; navigates to the slide specified by *newSlide*. For the navigation to succeed, the following must be true:

- The current slide must be a child slide of *mySlide*.
- The slide specified by *newSlide* and the current slide must share a common ancestor slide—that is, the current slide and *newSlide* must reside in the same slide subtree.

If either of these conditions isn't met, the navigation fails and the method returns `false`; otherwise, the method navigates to the specified slide and returns `true`.

For example, consider the following slide hierarchy:

```
Presentation
  Slide1
    Slide1_1
    Slide1_2
 Slide2
    Slide2_1
    Slide2_2
```

If the current slide is `Slide1_2`, the following `gotoSlide()` call will fail, since the current slide is not a descendant of `Slide2`:

```
Slide2.gotoSlide(Slide2_1);
```

Also consider the following screen hierarchy, where a form object is the parent screen of two separate slide trees:

```
Form_1
  Slide1
    Slide1_1
    Slide1_2
  Slide2
    Slide2_1
    Slide2_2
```

If the current slide is `Slide1_2`, the following method call will also fail, because `Slide1` and `Slide2` are in different slide subtrees:

```
Slide1_2.gotoSlide(Slide2_2);
```

Example

The following code, attached to a Button component, uses the `Slide.currentSlide` property and the `gotoSlide()` method to display the next slide in the presentation.

```
on(click) {
  _parent.gotoSlide(_parent.currentSlide.nextSlide);
}
```

This is equivalent to the following code, which uses the `Slide.gotoNextSlide()` method:

```
on(click) {
  _parent.currentSlide.gotoNextSlide();
}
```

See also

`Slide.currentSlide`, `Slide.gotoNextSlide()`

Slide.hideChild

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(hideChild) {
  // your code here
}
```

Description

Event; broadcast each time a child of a slide changes from visible to invisible. This event is broadcast only by slides, not forms. The main use of the `hideChild` event is to apply "out" transitions to all the children of a slide.

Example

When attached to the root slide (for example, the presentation slide), this code displays the name of each child slide that belongs to the root slide, as the child slide is hidden.

```
on(hideChild) {
  var child = eventObj.target._name;
  trace(child + " has just been hidden");
}
```

See also

`Slide.revealChild`

Slide.indexInParentSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.indexInParent

Description

Property (read-only); returns the zero-based index of *mySlide* in its parent's list of child slides.

Example

The following code uses the indexInParentSlide and Slide.numChildSlides properties to display the index of the current slide being viewed and the total number of slides contained by its parent slide. To use this code, attach it to a parent slide that contains one or more child slides.

```
on (revealChild) {
  trace("Displaying "+(currentSlide.indexInParentSlide+1)+" of
  "+currentSlide._parent.numChildSlides);
}
```

Note that because this property is a zero-based index, its value is incremented by 1 (currentSlide.indexInParent+1) to display more meaningful values.

See also

Slide.numChildSlides, Slide.revealChild

Slide.lastSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.lastSlide

Description

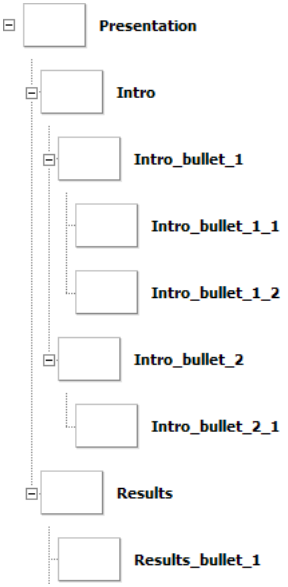
Property (read-only); returns the last child slide of *mySlide* that has no child slides.

Example

The following statements are all true concerning the slide hierarchy shown below:

```
Presentation.lastSlide._name == Results_bullet_1;
Intro.lastSlide._name == Intro_bullet_1_2;
```

```
Intro_bullet_1.lastSlide._name == Intro_bullet_1_2;
Results.lastSlide._name = Results_bullet_1;
```



Slide.nextSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.nextSlide

Description

Property (read-only); returns the slide you would reach if you called *mySlide*.gotoNextSlide(), but does not actually navigate to that slide. For example, you can use this property to display the name of the next slide in a presentation and let users select whether they want to navigate to that slide.

Example

In this example, the label of a Button component named nextButton displays the name of the next slide in the presentation. If there is no next slide—that is, if mySlide.nextSlide is null—then the button's label is updated to indicate that the user is at the end of this slide presentation.

```
if (mySlide.nextSlide != null) {
    nextButton.label = "Next slide: " + mySlide.nextSlide._name + " > ";
} else {
```

```
  nextButton.label = "End of this slide presentation.";
}
```

See also

Slide.gotoNextSlide(), Slide.previousSlide

Slide.numChildSlides

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.numChildSlides

Description

Property (read-only); returns the number of child slides that *mySlide* contains. A slide can contain either forms or other slides; if *mySlide* contains both slides and forms, this property only returns the number of slides, and does not count forms.

Example

This example uses Slide.numChildSlides and the Slide.getChildSlide() method to iterate over all the child slides of the root presentation slide. It then displays their names in the Output panel.

```
var numSlides = _root.Presentation.numChildSlides;
for(var slideIndex=0; slideIndex < numSlides; slideIndex++) {
  var childSlide = _root.Presentation.getChildSlide(slideIndex);
  trace(childSlide._name);
}
```

See also

Slide.getChildSlide()

Slide.overlayChildren

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.overlayChildren

Description

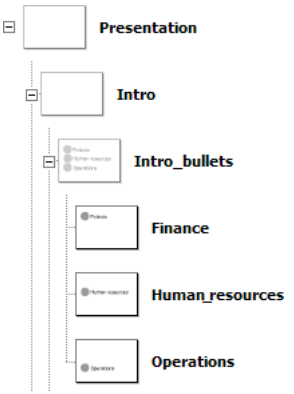
Property; determines whether child slides of *mySlide* remain visible when navigating from one child slide to the next. When this property is `true`, the previous slide remains visible when control passes to its next sibling slide; when this property is `false`, the previous slide is invisible when control passes to its next sibling slide.

Setting this property to `true` is useful, for example, when a given slide contains several child "bullet point" slides that are revealed separately (using transitions, perhaps), but all need to remain visible as new bullet points appear.

Note: This property applies only to the immediate descendants of *mySlide*, not to all (nested) child slides.

Example

The Intro_bullets slide in the following illustration contains three child slides (Finance, Human_resources, and Operations) that each display a separate bullet point. By setting `Intro_bullets.overlayChildren` to `true`, each bullet slide remains on the Stage as the other bullet points appear.



Slide.parentIsSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide`.parentIsSlide`

Description

Property (read-only); a Boolean value indicating whether the parent object of *mySlide* is also a slide. If the parent object of *mySlide* is a slide, or belongs to a subclass of Slide, this property returns `true`; otherwise, it returns `false`.

If *mySlide* is the root slide in a presentation, this property returns `false`, because the presentation slide's parent is the main Timeline (_level0), not a slide. This property also returns `false` if a form is the parent of *mySlide*.

Example

The following code determines whether the parent object of the slide `mySlide` is itself a slide. If `mySlide.parentIsSlide` is `true`, the number of `mySlide`'s sibling slides is displayed in the Output panel. If the parent object is not a slide, Flash assumes that `mySlide` is the root (master) slide in the presentation and therefore has no sibling slides.

```
if (mySlide.parentIsSlide) {
  trace("I have " + mySlide._parent.numChildSlides+" sibling slides");
} else {
  trace("I am the root slide and have no siblings");
}
```

See also

`Slide.numChildSlides`

Slide.playHidden

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide`.playHidden`

Description

Property; a Boolean value that specifies whether *mySlide* should continue to play when it is hidden. When this property is `true`, *mySlide* continues to play when hidden. When set to `false`, *mySlide* is stopped upon being hidden; upon being revealed, play restarts at Frame 1 of *mySlide*.

Slide.previousSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide`.previousSlide`

Description

Property (read-only); returns the slide you would reach if you called *mySlide*.gotoPreviousSlide(), but does not actually navigate to that slide. For example, you can use this property to display the name of the previous slide in a presentation and let users select whether they want to navigate to that slide.

Example

In this example, the label of a Button component named previousButton displays the name of the previous slide in the presentation. If there is no previous slide—that is, if mySlide.previousSlide is null—the button's label is updated to indicate that the user is at the beginning of this slide presentation.

```
if (mySlide.previousSlide != null) {
     previousButton.label = "Previous slide: " + mySlide.previous._name + " >
  ";
} else {
     previousButton.label = "You're at the beginning of this slide
  presentation.";
}
```

See also

Slide.gotoPreviousSlide(), Slide.nextSlide

Slide.revealChild

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
on(revealChild) {
  // your code here
}
```

Description

Event; broadcast each time a child slide of a slide object changes from invisible to visible. This event is used primarily to attach "in" transitions to all the child slides of a given slide.

Example

When attached to the root slide (for example, the presentation slide), this code will display the name of each child slide as it appears.

```
on(revealChild) {
  var child = eventObj.target._name;
  trace(child + " has just appeared");
}
```

See also

Slide.hideChild

Slide.rootSlide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

mySlide.rootSlide

Description

Property (read-only); returns the root slide of the slide tree, or slide subtree, that contains *mySlide*.

Example

Suppose you have a movie clip on a slide that, when clicked, goes to the first slide in the presentation. To accomplish this, you would attach the following code to the movie clip:

```
on(press) {
  _parent.rootSlide.gotoFirstSlide();
}
```

In this case, _parent refers to the slide that contains the movie clip object.

StyleManager class

ActionScript Class Name mx.styles.StyleManager

The StyleManager class keeps track of known inheriting styles and colors. You only need to use this class if you are creating components and want to add a new inheriting style or color.

To determine which styles are inheriting, see the W3C web site at www.w3.org/Style/CSS/.

Method summary for the StyleManager class

The following table lists methods of the StyleManager class.

Method	Description
`StyleManager.registerColorName()`	Registers a new color name with the Style Manager.
`StyleManager.registerColorStyle()`	Adds a new color style to the Style Manager.
`StyleManager.registerInheritingStyle()`	Registers a new inheriting style with the Style Manager.

StyleManager.registerColorName()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
StyleManager.registerColorName(colorName, value)
```

Parameters

colorName A string indicating the name of the color (for example, `"gray"`, `"darkGrey"`, and so on).

value A hexadecimal number indicating the color (for example, 0x808080, 0x404040, and so on).

Returns

Nothing.

Description

Method; associates a color name with a hexadecimal value and registers it with the Style Manager.

Example

The following example registers `"gray"` as the color name for the color represented by the hexadecimal value 0x808080:

```
StyleManager.registerColorName("gray", 0x808080);
```

StyleManager.registerColorStyle()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
StyleManager.registerColorStyle(colorStyle)
```

Parameters

colorStyle A string indicating the name of the color (for example, `"highlightColor"`, `"shadowColor"`, `"disabledColor"`, and so on).

Returns

Nothing.

Description

Method; adds a new color style to the Style Manager.

Example

The following example registers `"highlightColor"` as a color style:

```
StyleManager.registerColorStyle("highlightColor");
```

StyleManager.registerInheritingStyle()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
StyleManager.registerInheritingStyle(propertyName)
```

Parameters

propertyName A string indicating the name of the style property (for example, `"newProp1"`, `"newProp2"`, and so on).

Returns

Nothing.

Description

Method; marks this style property as inheriting. Use this method to register style properties that aren't listed in the CSS specification. Do not use this method to change non-inheriting style properties to inheriting.

When a style's value is not inherited, you can set its style only on an instance, not on a custom or global style sheet. A style that doesn't inherit its value is set on the class style sheet, and therefore, setting it on a custom or global style sheet does not work.

Example

The following example registers `newProp1` as an inheriting style:

```
StyleManager.registerInheritingStyle("newProp1");
```

SystemManager class

ActionScript Class Name mx.managers.SystemManager

The SystemManager class works automatically with the FocusManager class to handle which top-level window is activated in an application that contains version 2 components. It also provides a `screen` property that allows components to access Stage coordinates.

Property summary for the SystemManager class

The following table lists the property of the SystemManager class.

Property	Description
`SystemManager.screen`	Read-only; an object containing the size and position of the Stage.

SystemManager.screen

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
SystemManager.screen
```

Description

Property; an object with `x`, `y`, `width`, and `height` properties that indicate the size and position of the Stage.

Example

If `Stage.align` is set to something other than `"LT"`, it is very difficult to know what coordinates are actually viewable.

For example, suppose you want to place a watermark icon in the lower right corner of the Stage (similar to the watermarks many television channels use). If the Stage is centered and much wider than the Stage size set up in the FLA file, the following code places the watermark offscreen:

```
Watermark.move(Stage.width - Watermark.width, Stage.height -
  Watermark.height);
```

However, the following code would work in all Stage alignments:

```
Watermark.move(SystemManager.screen.width + SystemManager.screen.x -
  Watermark.width, SystemManager.screen.height + SystemManager.screen.x -
  Watermark.height);
```

TextArea component

The TextArea component wraps the native ActionScript TextField object. You can use styles to customize the TextArea component; when an instance is disabled, its contents display in a color represented by the disabledColor style. A TextArea component can also be formatted with HTML, or as a password field that disguises the text. See "Applying a style sheet to a TextArea component" in *Using ActionScript in Flash*.

A TextArea component can be enabled or disabled in an application. In the disabled state, it doesn't receive mouse or keyboard input. When enabled, it follows the same focus, selection, and navigation rules as an ActionScript TextField object. When a TextArea instance has focus, you can use the following keys to control it:

Key	Description
Arrow keys	Move the insertion point one line up, down, left, or right.
Page Down	Moves one screen down.
Page Up	Moves one screen up.
Shift+Tab	Moves focus to the previous object.
Tab	Moves focus to the next object.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each TextArea instance reflects changes made to parameters in the Property inspector or Component inspector during authoring. If a scroll bar is needed, it appears in the live preview, but it does not function. Text is not selectable in the live preview, and you cannot enter text in the component instance on the Stage.

When you add the TextArea component to an application, you can use the Accessibility panel to make it accessible to screen readers.

Using the TextArea component

You can use a TextArea component wherever you need a multiline text field. If you need a single-line text field, use the TextInput component. For example, you could use a TextArea component as a comment field in a form. You could set up a listener that checks if the field is empty when a user tabs out of the field. That listener could display an error message indicating that a comment must be entered in the field.

TextArea parameters

You can set the following authoring parameters for each TextArea component instance in the Property inspector or in the Component inspector:

text indicates the contents of the TextArea component. You cannot enter carriage returns in the Property inspector or Component inspector. The default value is "" (an empty string).

html indicates whether the text is formatted with HTML (`true`) or not (`false`). The default value is `false`.

editable indicates whether the TextArea component is editable (`true`) or not (`false`). The default value is `true`.

wordWrap indicates whether the text wraps (`true`) or not (`false`). The default value is `true`.

You can write ActionScript to control these and additional options for the TextArea component using its properties, methods, and events. For more information, see "TextArea class" on page 728.

Creating an application with the TextArea component

The following procedure explains how to add a TextArea component to an application while authoring. In this example, the component is a Comment field with an event listener that determines if a user has entered text.

To create an application with the TextArea component:

1. Drag a TextArea component from the Components panel to the Stage.
2. In the Property inspector, enter the instance name **comment**.
3. In the Property inspector, set parameters as you wish. However, leave the text parameter blank, the editable parameter set to `true`, and the password parameter set to `false`.
4. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
textListener = new Object();
textListener.handleEvent = function (evt){
  if (comment.length < 1)  {
  Alert(_root, "Error", "You must enter at least a comment in this field",
  mxModal | mxOK);
  }
}
comment.addEventListener("focusOut", textListener);
```

   This code sets up a `focusOut` event handler on the TextArea comment instance that verifies that the user typed something in the text field.
5. Once text is entered in the comment instance, you can get its value as follows:

```
var login = comment.text;
```

Customizing the TextArea component

You can transform a TextArea component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use `UIObject.setSize()` or any applicable properties and methods of the TextArea class.

When a TextArea component is resized, the border is resized to the new bounding box. The scroll bars are placed on the bottom and right edges if they are required. The text field is then resized within the remaining area; there are no fixed-size elements in a TextArea component. If the TextArea component is too small to display the text, the text is clipped.

Using styles with the TextArea component

The TextArea component supports one set of component styles for all text in the field. However, you can also display HTML that is compatible with Flash Player HTML rendering. To display HTML text, set `TextArea.html` to `true`.

The TextArea component has its `backgroundColor` and `borderStyle` style properties defined on a class style declaration. Class styles override global styles; therefore, if you want to set the `backgroundColor` and `borderStyle` style properties, you must create a different custom style declaration on the instance.

If the name of a style property ends in "Color", it is a color style property and behaves differently than noncolor style properties. For more information, see "Using styles to customize component color and text" on page 67.

A TextArea component supports the following styles:

Style	Theme	Description
`backgroundColor`	Both	The background color. The default color is white.
`border styles`	Both	The TextArea component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style is `"inset"`.
`marginLeft`	Both	A number indicating the left margin for text. The default value is 0.
`marginRight`	Both	A number indicating the right margin for text. The default value is 0.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.

Style	Theme	Description
textAlign	Both	The text alignment: either "left", "right", or "center". The default value is "left".
textIndent	Both	A number indicating the text indent. The default value is 0.
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".

The TextArea and TextInput components use exactly the same styles and are often used in the same manner. Thus, by default they share the same class-level style declaration. For example, the following code sets a style on the TextInput declaration, but it affects both TextInput and TextArea components.

```
_global.styles.TextInput.setStyle("disabledColor", 0xBBBBFF);
```

To separate the components and provide class-level styles for one and not the other, create a new style declaration.

```
import mx.styles.CSSStyleDeclaration;
_global.styles.TextArea = new CSSStyleDeclaration();
_global.styles.TextArea.setStyle("disabledColor", 0xFFBBBB);
```

This example does not check if `_global.styles.TextArea` existed before overwriting it; it assumes you know it exists and want to overwrite it.

Using skins with the TextArea component

The TextArea component uses an instance of RectBorder for its border and scroll bars for scrolling images. For more information about skinning these visual elements, see "RectBorder class" on page 647 and "Using skins with the UIScrollBar component" on page 831.

TextArea class

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > TextArea

ActionScript Class Name mx.controls.TextArea

The properties of the TextArea class let you set the text content, formatting, and horizontal and vertical position at runtime. You can also indicate whether the field is editable, and whether it is a "password" field. You can also restrict the characters that a user can enter.

Setting a property of the TextArea class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

The TextArea component overrides the default Flash Player focus rectangle and draws a custom focus rectangle with rounded corners.

The TextArea component supports CSS styles and any additional HTML styles supported by Flash Player.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.TextArea.version);
```

Note: The code `trace(myTextAreaInstance.version);` returns `undefined`.

Method summary for the TextArea class

There are no methods exclusive to the TextArea class.

Methods inherited from the UIObject class

The following table lists the methods the TextArea class inherits from the UIObject class. When calling these methods from the TextArea object, use the form *`TextAreaInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.
`UIObject.setStyle()`	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the TextArea class inherits from the UIComponent class. When calling these methods from the TextArea object, use the form *`TextAreaInstance.methodName`*.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Property summary for the TextArea class

The following table lists properties of the TextArea class.

Property	Description
TextArea.editable	A Boolean value indicating whether the field is editable (true) or not (false).
TextArea.hPosition	Defines the horizontal position of the text in the field.
TextArea.hScrollPolicy	Indicates whether the horizontal scroll bar is always on ("on"), is never on ("off"), or turns on when needed ("auto").
TextArea.html	A Boolean value that indicates whether the text field can be formatted with HTML.
TextArea.length	Read-only; the number of characters in the text field.
TextArea.maxChars	The maximum number of characters that the text field can contain.
TextArea.maxHPosition	Read-only; the maximum value of TextArea.hPosition.
TextArea.maxVPosition	Read-only; the maximum value of TextArea.vPosition.
TextArea.password	A Boolean value indicating whether the field is a password field (true) or not (false).
TextArea.restrict	The set of characters that a user can enter in the text field.
TextArea.styleSheet	Attaches a style sheet to the specified TextArea component.
TextArea.text	The text contents of a TextArea component.
TextArea.vPosition	A number indicating the vertical scrolling position.
TextArea.vScrollPolicy	Indicates whether the vertical scroll bar is always on ("on"), is never on ("off"), or turns on when needed ("auto").
TextArea.wordWrap	A Boolean value indicating whether the text wraps (true) or not (false).

Properties inherited from the UIObject class

The following table lists the properties the TextArea class inherits from the UIObject class. When accessing these properties from the TextArea object, use the form *TextAreaInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.

Property	Description
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the TextArea class inherits from the UIComponent class. When accessing these properties from the TextArea object, use the form *TextAreaInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the TextArea class

The following table lists the event of the TextArea class.

Event	Description
TextArea.change	Notifies listeners that text has changed.

Events inherited from the UIObject class

The following table lists the events the TextArea class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the TextArea class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

TextArea.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(change){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  ...
}
textAreaInstance.addEventListener("change", listenerObject)
```

Description

Event; notifies listeners that text has changed. This event is broadcast after the text has changed. This event cannot be used to prevent certain characters from being added to the component's text field; for this purpose, use `TextArea.restrict`.

The first usage example uses an `on()` handler and must be attached directly to a TextArea instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the instance `myTextArea`, sends "_level0.myTextArea" to the Output panel:

```
on(change){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*textAreaInstance*) dispatches an event (in this case, `change`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example uses the dispatcher/listener event model to track the total of number of times the text field has changed in a TextArea component named `myTextArea`:

```
// define a listener object
var myTextAreaListener:Object = new Object();

// create a Number variable to track the number of changes to the TextArea
var changeCount:Number = 0;

// define a function that is executed whenever the listener receives
// notification of a change in the TextArea component
myTextAreaListener.change = function(eventObj) {
  changeCount++;
  trace("Text has changed " + changeCount + " times now!");
  trace("It now contains: " + eventObj.target.text);
}

// register the listener object with the TextArea component instance
myTextArea.addEventListener("change", myTextAreaListener);
```

See also

`EventDispatcher.addEventListener()`

TextArea.editable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.editable

Description

Property; a Boolean value that indicates whether the component is editable (`true`) or not (`false`). The default value is `true`.

TextArea.hPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`textAreaInstance``.hPosition`

Description

Property; defines the horizontal position of the text in the field. The default value is 0.

Example

The following code displays the leftmost characters in the field:

```
myTextArea.hPosition = 0;
```

TextArea.hScrollPolicy

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`textAreaInstance``.hScrollPolicy`

Description

Property; determines whether the horizontal scroll bar is always present (`"on"`), is never present (`"off"`), or appears automatically according to the size of the field (`"auto"`). The default value is `"auto"`.

Example

The following code turns horizontal scroll bars on all the time:

```
text.hScrollPolicy = "on";
```

TextArea.html

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.html

Description

Property; a Boolean value that indicates whether the text field is formatted with HTML (true) or not (false). If the html property is true, the text field is an HTML text field. If html is false, the text field is a non-HTML text field. The default value is false.

Example

The following example makes the myTextArea field an HTML text field and then formats the text with HTML tags:

```
myTextArea.html = true;
myTextArea.text = "The <b>Royal</b> Nonesuch"; // displays "The Royal
  Nonesuch"
```

TextArea.length

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.length

Description

Property (read-only); indicates the number of characters in a text field. This property returns the same value as the ActionScript text.length property, but is faster. A character such as tab ("\t") counts as one character. The default value is 0.

Example

The following example gets the length of the text field and copies it to the length variable:

```
var length = myTextArea.length; // find out how long the text string is
```

TextArea.maxChars

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.maxChars

Description

Property; the maximum number of characters that the text field can contain. A script may insert more text than the `maxChars` property allows; the property indicates only how much text a user can enter. If the value of this property is `null`, there is no limit to the amount of text a user can enter. The default value is `null`.

Example

The following example limits the number of characters a user can enter to 255:

```
myTextArea.maxChars = 255;
```

TextArea.maxHPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.maxHPosition

Description

Property (read-only); the maximum value of `TextArea.hPosition`. The default value is 0.

Example

The following code causes the text to scroll to the far right:

```
myTextArea.hPosition = myTextArea.maxHPosition;
```

See also

`TextArea.vPosition`

TextArea.maxVPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.maxVPosition

Description

Property (read-only); indicates the maximum value of TextArea.vPosition. The default value is 0.

Example

The following code causes the text to scroll to the bottom of the component:

```
myTextArea.vPosition = myTextArea.maxVPosition;
```

See also

TextArea.hPosition

TextArea.password

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.password

Description

Property; a Boolean value indicating whether the text field is a password field (true) or not (false). If password is true, the text field is a password text field and hides the input characters. If password is false, the text field is not a password text field. The default value is false.

Example

The following code makes the text field a password field that displays all characters as asterisks (*):

```
myTextArea.password = true;
```

TextArea.restrict

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.restrict

Description

Property; indicates the set of characters that users can enter in the text field. The default value is `undefined`. If this property is `null`, users can enter any character. If this property is an empty string, no characters can be entered. If this property is a string of characters, users can enter only characters in the string; the string is scanned from left to right. You can specify a range by using a dash (-).

If the string begins with ^, all characters that follow the ^ are considered unacceptable characters. If the string does not begin with ^, the characters in the string are considered acceptable. The ^ can also be used as a toggle between acceptable and unacceptable characters.

For example, the following code allows A-Z except X and Q:

```
Ta.restrict = "A-Z^XQ";
```

The `restrict` property only restricts user interaction; a script may put any text into the text field. This property does not synchronize with the Embed Font Outlines check boxes in the Property inspector.

Example

In the following example, the first line of code limits the text field to uppercase letters, numbers, and spaces. The second line of code allows all characters except lowercase letters.

```
my_txt.restrict = "A-Z 0-9"; // limit control to uppercase letters, numbers,
  and spaces
my_txt.restrict = "^a-z"; // allow all characters, except lowercase letters
```

TextArea.styleSheet

Availability

Flash Player 7.

Usage

TextAreaInstance.styleSheet = *TextFieldStyleSheetObject*

Description

Property; attaches a style sheet to the TextArea component specified by *TextAreaInstance*. For information on creating style sheets, see "Formatting text with Cascading Style Sheets" in *Using ActionScript in Flash*.

The style sheet associated with a TextArea component may be changed at any time. If the style sheet in use is changed, the TextArea component is redrawn with the new style sheet. The style sheet may be set to `null` or `undefined` to remove the style sheet. If the style sheet in use is removed, the TextArea component is redrawn without a style sheet. The formatting done by a style sheet is not retained if the style sheet is removed.

Example

The following code creates a new StyleSheet object named `styles` with the `new TextField.StyleSheet` constructor. It then defines styles for `<html>`, `<body>` and `<td>` tags. It then uses `LoadVars.load` to load an HTML file named myText.htm. That file contains text within `<html>`, `<body>` and `<td>` tags. When the text is displayed in the TextArea instance `MyTextArea`, the text within those tags is styled according to the StyleSheet object `styles`.

```
// create the new StyleSheet object
var styles = new TextField.StyleSheet();

// define styles for <html>, <body>, and <td>...
styles.setStyle("html",
  {fontFamily: 'Arial,Helvetica,sans-serif',
  fontSize: '12px',
  color: '#0000FF'});

styles.setStyle("body",
  {color: '#00CCFF',
  textDecoration: 'underline'});

styles.setStyle("td",
  {fontFamily: 'Arial,Helvetica,sans-serif',
  fontSize: '24px',
  color: '#006600'});

// set the TextAreaInstance.styleSheet property to the newly defined
// styleSheet object named styles
myTextArea.styleSheet=styles;
myTextArea.html=true;

// Load text to display and define onLoad handler
var myVars:LoadVars = new LoadVars();
myVars.load("myText.htm");
myVars.onData = function(content) {
  _root.myTextArea.text = content;
};
```

TextArea.text

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.text

Description

Property; the text contents of a TextArea component. The default value is "" (an empty string).

Example

The following code places a string in the myTextArea instance, and then traces that string to the Output panel:

```
myTextArea.text = "The Royal Nonesuch";
trace(myTextArea.text); // traces "The Royal Nonesuch"
```

TextArea.vPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.vPosition

Description

Property; defines the vertical scroll position of text in a text field. This property is useful for directing users to a specific paragraph in a long passage, or creating scrolling text fields. You can get and set this property. The default value is 0.

Example

The following code displays the topmost characters in a field:

```
myTextArea.vPosition = 0;
```

TextArea.vScrollPolicy

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.vScrollPolicy

Description

Property; determines whether the vertical scroll bar is always present ("on"), is never present ("off"), or appears automatically according to the size of the field ("auto"). The default value is "auto".

Example

The following code turns vertical scroll bars off all the time:

```
text.vScrollPolicy = "off";
```

TextArea.wordWrap

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textAreaInstance.wordWrap

Description

Property; a Boolean value that indicates whether the text wraps (`true`) or not (`false`). The default value is `true`.

TextInput component

The TextInput component is a single-line text component that is a wrapper for the native ActionScript TextField object. You can use styles to customize the TextInput component; when an instance is disabled, its contents appear in a color represented by the disabledColor style. A TextInput component can also be formatted with HTML, or as a password field that disguises the text.

A TextInput component can be enabled or disabled in an application. In the disabled state, it doesn't receive mouse or keyboard input. When enabled, it follows the same focus, selection, and navigation rules as an ActionScript TextField object. When a TextInput instance has focus, you can also use the following keys to control it:

Key	Description
Arrow keys	Move the insertion point one character left and right.
Shift+Tab	Moves focus to the previous object.
Tab	Moves focus to the next object.

For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

A live preview of each TextInput instance reflects changes made to parameters in the Property inspector or Component inspector during authoring. Text is not selectable in the live preview, and you cannot enter text in the component instance on the Stage.

When you add the TextInput component to an application, you can use the Accessibility panel to make it accessible to screen readers.

Using the TextInput component

You can use a TextInput component wherever you need a single-line text field. If you need a multiline text field, use the TextArea component. For example, you could use a TextInput component as a password field in a form. You could also set up a listener that checks if the field has enough characters when a user tabs out of the field. That listener could display an error message indicating that the proper number of characters must be entered.

TextInput parameters

You can set the following authoring parameters for each TextInput component instance in the Property inspector or in the Component inspector:

text specifies the contents of the TextInput component. You cannot enter carriage returns in the Property inspector or Component inspector. The default value is "" (an empty string).

editable indicates whether the TextInput component is editable (`true`) or not (`false`). The default value is `true`.

password indicates whether the field is a password field (`true`) or not (`false`). The default value is `false`.

You can write ActionScript to control these and additional options for the TextInput component using its properties, methods, and events. For more information, see "TextInput class" on page 745.

Creating an application with the TextInput component

The following procedure explains how to add a TextInput component to an application while authoring. In this example, the component is a password field with an event listener that determines if the proper number of characters has been entered.

To create an application with the TextInput component:

1. Drag a TextInput component from the Components panel to the Stage.
2. In the Property inspector, do the following:
   - Enter the instance name **passwordField**.
   - Leave the text parameter blank.
   - Set the editable parameter to `true`.
   - Set the password parameter to `true`.
3. Select Frame 1 in the Timeline, open the Actions panel, and enter the following code:

```
textListener = new Object();
textListener.handleEvent = function (evt){
  if (evt.type == "enter"){
    trace("You must enter at least 8 characters");
  }
}
passwordField.addEventListener("enter", textListener);
```

   This code sets up an `enter` event handler on the TextInput passwordField instance that verifies that the user entered the proper number of characters.
4. Once text is entered in the passwordField instance, you can get its value as follows:

```
var login = passwordField.text;
```

Customizing the TextInput component

You can transform a TextInput component horizontally while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use `UIObject.setSize()` or any applicable properties and methods of the TextInput class.

When a TextInput component is resized, the border is resized to the new bounding box. The TextInput component doesn't use scroll bars, but the insertion point scrolls automatically as the user interacts with the text. The text field is then resized within the remaining area; there are no fixed-size elements in a TextInput component. If the TextInput component is too small to display the text, the text is clipped.

Using styles with the TextInput component

The TextInput component has its `backgroundColor` and `borderStyle` style properties defined on a class style declaration. Class styles override global styles; therefore, if you want to set the `backgroundColor` and `borderStyle` style properties, you must create a different custom style declaration or define it on the instance.

A TextInput component supports the following styles:

Style	Theme	Description
`backgroundColor`		The background color. The default color is white.
border styles	Both	The TextArea component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style is `"inset"`.
`marginLeft`	Both	A number indicating the left margin for text. The default value is 0.
`marginRight`	Both	A number indicating the right margin for text. The default value is 0.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is `10`.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textAlign`	Both	The text alignment: either `"left"`, `"right"`, or `"center"`. The default value is `"left"`.
`textIndent`	Both	A number indicating the text indent. The default value is 0.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.

The TextArea and TextInput components both use the same styles and are often used in the same manner. Thus, by default they share the same class-level style declaration. For example, the following code sets a style on the TextArea declaration but it affects both TextArea and TextInput components.

```
_global.styles.TextArea.setStyle("disabledColor", 0xBBBBFF);
```

To separate the components and provide class-level styles for one and not the other, create a new style declaration.

```
import mx.styles.CSSStyleDeclaration;
_global.styles.TextInput = new CSSStyleDeclaration();
_global.styles.TextInput.setStyle("disabledColor", 0xFFBBBB);
```

Notice how this example does not check if `_global.styles.TextInput` existed before overwriting it; in this example, you know it exists and you want to overwrite it.

Using skins with the TextInput component

The TextArea component uses an instance of RectBorder for its border. For more information about skinning these visual elements, see "RectBorder class" on page 647.

TextInput class

Inheritance MovieClip > UIObject class > UIComponent class > TextInput

ActionScript Class Name mx.controls.TextInput

The properties of the TextInput class let you set the text content, formatting, and horizontal position at runtime. You can also indicate whether the field is editable, and whether it is a "password" field. You can also restrict the characters that a user can enter.

Setting a property of the TextInput class with ActionScript overrides the parameter of the same name set in the Property inspector or Component inspector.

The TextInput component uses the Focus Manager to override the default Flash Player focus rectangle and draw a custom focus rectangle with rounded corners. For more information, see "FocusManager class" on page 419.

The TextInput component supports CSS styles and any additional HTML styles supported by Flash Player. For information about CSS support, see the W3C specification at www.w3.org/TR/REC-CSS2/.

You can manipulate the text string by using the string returned by the text object.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.TextInput.version);
```

Note: The code `trace(myTextInputInstance.version);` returns `undefined`.

Method summary for the TextInput class

There are no methods exclusive to the TextInput class.

Methods inherited from the UIObject class

The following table lists the methods the TextInput class inherits from the UIObject class. When calling these methods from the TextInput object, use the form *TextInputInstance.methodName*.

Method	Description
UIObject.createClassObject()	Creates an object on the specified class.
UIObject.createObject()	Creates a subobject on an object.
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the TextInput class inherits from the UIComponent class. When calling these methods from the TextInput object, use the form *TextInputInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the TextInput class

The following table lists properties of the TextInput class.

Property	Description
TextInput.editable	A Boolean value indicating whether the field is editable (`true`) or not (`false`).
TextInput.hPosition	The horizontal scrolling position of the text in the field.

Property	Description
TextInput.length	The number of characters in a TextInput component. This property is read-only.
TextInput.maxChars	The maximum number of characters that a user can enter in the text field.
TextInput.maxHPosition	The maximum possible value for TextField.hPosition. This property is read-only.
TextInput.password	A Boolean value that indicates whether the text field is a password field that hides the entered characters.
TextInput.restrict	Indicates which characters a user can enter in a text field.
TextInput.text	Sets the text content of a TextInput component.

Properties inherited from the UIObject class

The following table lists the properties the TextInput class inherits from the UIObject class. When accessing these properties from the TextInput object, use the form *TextInputInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (true) or not (false).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the TextInput class inherits from the UIComponent class. When accessing these properties from the TextInput object, use the form *TextInputInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the TextInput class

The following table lists events of the TextInput class.

Event	Description
TextInput.change	Broadcast when the TextInput field changes.
TextInput.enter	Broadcast when the Enter key is pressed.

Events inherited from the UIObject class

The following table lists the events the TextInput class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the TextInput class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

TextInput.change

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(change){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.change = function(eventObject){
  ...
}
textInputInstance.addEventListener("change", listenerObject)
```

Description

Event; notifies listeners that text has changed. This event is broadcast after the text has changed. This event cannot be used to prevent certain characters from being added to the component's text field; for that purpose, use `TextInput.restrict`. This event is triggered only by user input, not by programmatic change.

The first usage example uses an `on()` handler and must be attached directly to a TextInput instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the instance `myTextInput`, sends "_level0.myTextInput" to the Output panel:

```
on(change){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`textInputInstance`*) dispatches an event (in this case, `change`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example sets a flag in the application that indicates if contents in the TextInput field have changed:

```
form.change = function(eventObj){
  // note: eventObj.target refers to the component that generated the change
  // event, i.e., the TextInput component.
  myFormChanged.visible = true;  // set a change indicator if the contents
  changed;
}
myInput.addEventListener("change", form);
```

See also

`EventDispatcher.addEventListener()`

TextInput.editable

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`textInputInstance``.editable`

Description

Property; a Boolean value that indicates whether the component is editable (`true`) or not (`false`). The default value is `true`.

TextInput.enter

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(enter){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.enter = function(eventObject){
  ...
```

```
}
textInputInstance.addEventListener("enter", listenerObject)
```

Description

Event; notifies listeners that the Enter key has been pressed.

The first usage example uses an `on()` handler and must be attached directly to a TextInput instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the instance `myTextInput`, sends "_level0.myTextInput" to the Output panel:

```
on(enter){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`textInputInstance`*) dispatches an event (in this case, `enter`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

This example sets a flag in the application that indicates if contents in the TextInput field have changed:

```
form.enter = function(eventObj){
  // note: eventObj.target refers the component that generated the enter event,
  // i.e., the TextInput component.
  myFormChanged.visible = true;
  // set a change indicator if the user presses Enter;
}
myInput.addEventListener("enter", form);
```

See also

`EventDispatcher.addEventListener()`

TextInput.hPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textInputInstance.hPosition

Description

Property; defines the horizontal position of the text in the field. The default value is 0.

Example

The following code displays the leftmost character in the field:

```
myTextInput.hPosition = 0;
```

TextInput.length

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

inputInstance.length

Description

Property (read-only); a number that indicates the number of characters in a TextInput component. A character such as tab (`"\t"`) counts as one character. The default value is 0.

Example

The following code determines the number of characters in the `myTextInput` string and copies it to the `length` variable:

```
var length = myTextInput.length;
```

TextInput.maxChars

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textInputInstance.maxChars

Description

Property; the maximum number of characters that the text field can contain. A script may insert more text than the `maxChars` property allows; this property indicates only how much text a user can enter. If this property is `null`, there is no limit to the amount of text a user can enter. The default value is `null`.

Example

The following example limits the number of characters a user can enter to 255:

```
myTextInput.maxChars = 255;
```

TextInput.maxHPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textInputInstance.maxHPosition

Description

Property (read-only); indicates the maximum value of `TextInput.hPosition`. The default value is 0.

Example

The following code scrolls to the far right:

```
myTextInput.hPosition = myTextInput.maxHPosition;
```

TextInput.password

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textInputInstance.password

Description

Property; a Boolean value indicating whether the text field is a password field (`true`) or not (`false`). If this property is `true`, the text field is a password text field and hides the input characters. If this property is `false`, the text field is not a password text field. The default value is `false`.

Example

The following code makes the text field a password field that displays all characters as asterisks (*):

```
myTextInput.password = true;
```

TextInput.restrict

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

textInputInstance.restrict

Description

Property; indicates the set of characters that a user can enter in the text field. The default value is `undefined`. If this property is `null` or an empty string (`""`), a user can enter any character. If this property is a string of characters, the user can enter only characters in the string; the string is scanned from left to right. You can specify a range by using a dash (-).

If the string begins with `^`, all characters that follow the `^` are considered unacceptable characters. If the string does not begin with `^`, the characters in the string are considered acceptable. The `^` can also be used as a toggle between acceptable and unacceptable characters.

For example, the following code allows A-Z except X and Q:

```
Ta.restrict = "A-Z^XQ";
```

You can use the backslash (`\`) to enter a hyphen (`-`), caret (`^`), or backslash (`\`) character, as shown here:

```
\^
\-
\\
```

When you enter the `\` character in the Actions panel within double quotation marks, it has a special meaning for the Actions panel's double-quote interpreter. It signifies that the character following the `\` should be treated as is. For example, you could use the following code to enter a single quotation mark:

```
var leftQuote = "\'";
```

The Actions panel's restrict interpreter also uses `\` as an escape character. Therefore, you may think that the following should work:

```
myText.restrict = "0-9\-\^\\";
```

However, since this expression is surrounded by double quotation marks, the following value is sent to the restrict interpreter: `0-9-^\`, and the restrict interpreter doesn't understand this value.

Because you must enter this expression within double quotation marks, you must not only provide the expression for the restrict interpreter, but you must also escape the Actions panel's built-in interpreter for double quotation marks. To send the value `0-9\-\^\\` to the restrict interpreter, you must enter the following code:

```
myText.restrict = "0-9\\-\\^\\\\";
```

The `restrict` property restricts only user interaction; a script may put any text into the text field. This property does not synchronize with the Embed Font Outlines check boxes in the Property inspector.

Example

In the following example, the first line of code limits the text field to uppercase letters, numbers, and spaces. The second line of code allows all characters except lowercase letters.

```
my_txt.restrict = "A-Z 0-9";
my_txt.restrict = "^a-z";
```

The following code allows a user to enter the characters "0 1 2 3 4 5 6 7 8 9 - ^ \" in the instance `myText`. You must use a double backslash to escape the characters `-`, `^`, and \. The first `\` escapes the double quotation marks, and the second `\` tells the interpreter that the next character should not be treated as a special character.

```
myText.restrict = "0-9\\-\\^\\\\";
```

TextInput.text

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`textInputInstance``.text`

Description

Property; the text contents of a TextInput component. The default value is `""` (an empty string).

Example

The following code places a string in the `myTextInput` instance, and then traces that string to the Output panel:

```
myTextInput.text = "The Royal Nonesuch";
trace(myTextInput.text); // traces "The Royal Nonesuch"
```

TransferObject interface

ActionScript Class Name mx.data.to.TransferObject

The TransferObject interface defines a set of methods that items managed by the DataSet component must implement. The `DataSet.itemClassName` property specifies the name of the transfer object class that is instantiated each time a new item is needed. You can also specify this property for a selected DataSet component using the Property inspector.

Method summary for the TransferObject interface

The following table lists methods of the TransferObject interface.

Method	Description
`TransferObject.clone()`	Creates a new instance of the transfer object.
`TransferObject.getPropertyData()`	Returns the data for this transfer object.
`TransferObject.setPropertyData()`	Sets the data for this transfer object.

TransferObject.clone()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
class itemClass implements mx.data.to.TransferObject {
  function clone() {
    // your code here
  }
}
```

Parameters

None.

Returns

A copy of the transfer object.

Description

Method; creates an instance of the transfer object. The implementation of this method creates a copy of the existing transfer object and its properties and then returns that object.

Example

The following function returns a copy of this transfer object with all of the properties set to the same values as the original:

```
class itemClass implements mx.data.to.TransferObject {
  function clone():Object {
    var copy:itemClass = new itemClass();
    for (var p in this) {
      copy[p]= this[p];
    }
    return(copy);
  }
}
```

TransferObject.getPropertyData()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
class itemClass implements mx.data.to.TransferObject {
  function getPropertyData() {
  // your code here
  }
}
```

Parameters

None.

Returns

An object.

Description

Method; returns the data for this transfer object. The implementation of this method can return an anonymous ActionScript object with properties and corresponding values.

Example

The following function returns an object named `internalData` that contains the properties and their values from the Contact object:

```
class Contact implements mx.data.to.TransferObject {
function getPropertyData():Object {
  var internalData:Object = {name:name, readOnly:_readOnly, phone:phone,
  zip:zip.zipPlus4};
  return(internalData);
}
```

TransferObject.setPropertyData()

Availability

Flash Player 7.

Edition

Flash MX 2004.

Usage

```
class yourClass implements TransferObject {
  function setPropertyData(propData) {
  // your code here
  }
}
```

Parameters

propData An object that contains the data assigned to this transfer object.

Returns

Nothing.

Description

Method; sets the data for this transfer object. The *propData* parameter is an object whose fields contain the data assigned by the DataSet component to this transfer object.

Example

The following function receives a *propData* parameter and applies the values of its properties to the properties of the Contact object:

```
class Contact implements mx.data.to.TransferObject {

  function setPropertyData(propData: Object):Void {
    _readOnly = propData.readOnly;
    phone = propData.phone;
    zip = new mx.data.types.ZipCode(data.zip);
  }

  public var name:String;
  public var phone:String;
  public var zip:ZipCode;
  private var _readOnly:Boolean; // indicates if immutable
}
```

Tree component (Flash Professional only)

The Tree component allows a user to view hierarchical data. The tree appears in a box like the List component, but each item in a tree is called a *node* and can be either a *leaf* or a *branch*. 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 an expander arrow (disclosure triangle) that a user can open to expose children. The children of a branch can be leaves or branches.

The data of a tree component must be provided from an XML data source. For more information, see "Using the Tree component (Flash Professional only)" on page 759.

When a Tree instance has focus either from clicking or tabbing, you can use the following keys to control it:

Key	Description
Down Arrow	Moves selection down one item.
Up Arrow	Moves selection up one item.
Right Arrow	Opens a selected branch node. If a branch is already open, moves to first child node.
Left Arrow	Closes a selected branch node. If on a leaf node of a closed branch node, moves to parent node.
Space	Opens or closes a selected branch node.
End	Moves selection to the bottom of the list.
Home	Moves selection to the top of the list.
Page Down	Moves selection down one page.
Page Up	Moves selection up one page.
Control	Allows multiple noncontiguous selections.
Shift	Allows multiple contiguous selections.

The Tree component cannot be made accessible to screen readers.

Using the Tree component (Flash Professional only)

The Tree component can be used to represent hierarchical data such as e-mail client folders, file browser panes, or category browsing systems for inventory. Most often, the data for a tree is retrieved from a server in the form of XML, but it can also be well-formed XML that is created during authoring in Flash. The best way to create XML for the tree is to use the TreeDataProvider interface. You can also use the ActionScript XML class or build an XML string. After you create an XML data source (or load one from an external source), you assign it to `Tree.dataProvider`.

The Tree component comprises two sets of APIs: the Tree class and the TreeDataProvider interface. The Tree class contains the visual configuration methods and properties. The TreeDataProvider interface lets you construct XML and add it to multiple tree instances. A TreeDataProvider object broadcasts changes to any trees that use it. In addition, any XML or XMLNode object that exists on the same frame as a tree or a menu is automatically given the TreeDataProvider methods and properties. For more information, see "TreeDataProvider interface (Flash Professional only)" on page 787.

Formatting XML for the Tree component

The Tree component is designed to display hierarchical data structures using XML as the data model. It is important to understand the relationship of the XML data source to the Tree component.

Consider the following XML data source sample:

```
<node>
  <node label="Mail">
       <node label="INBOX"/>
       <node label="Personal Folder">
           <node label="Business" isBranch="true" />
           <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>
```

Note: The `isBranch` attribute is read-only; you cannot set it directly. To set it, call `Tree.setIsBranch()`.

Nodes in the XML data source can have any name. Notice in the previous example that each node is named with the generic name `node`. The tree reads through the XML and builds the display hierarchy according to the nested relationship of the nodes.

Each XML node can be displayed as one of two types in the tree: branch or leaf. Branch nodes can contain multiple child nodes and appear as a folder icon with an expander arrow that allows users to open and close the folder. Leaf nodes appear as a file icon and cannot contain child nodes. Both leaves and branches can be roots; a root node appears at the top level of the tree and has no parent. The icons are customizable; for more information, see "Using skins with the Tree component" on page 769.

There are many ways to structure XML, but the Tree component cannot use all types of XML structures. Do not nest node attributes in a child node; each node should contain all its necessary attributes. Also, the attributes of each node should be consistent to be useful. For example, to describe a mailbox structure with a Tree component, use the same attributes on each node (message, data, time, attachments, and so on). This lets the tree know what it expects to render, and lets you loop through the hierarchy to compare data.

When a Tree displays a node, it uses the `label` attribute of the node by default as the text label. If any other attributes exist, they become additional properties of the node's attributes within the tree.

The actual root node is interpreted as the Tree component itself. This means that the first child (in the previous example, `<node label="Mail">`), is rendered as the root node in the tree view. This means that a tree can have multiple root folders. In the example, there is only one root folder displayed in the tree: Mail. However, if you were to add sibling nodes at that level in the XML, multiple root nodes would be displayed in the tree.

A data provider for a tree always wants a node that has children it can display. It displays the first child of the XMLNode object. When the XML is wrapped in an XML object, the structure looks like the following:

```
<XMLDocumentObject>
  <node>
    <node label="Mail">
      <node label="INBOX"/>
      <node label="Personal Folder">
        <node label="Business" isBranch="true" />
        <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>
</XMLDocumentObject>
```

Flash Player wraps the XML nodes in an extra document node, which is passed to the tree. When the tree tries to display the XML, it tries to display `<node>`, which doesn't have a label, so it doesn't display correctly.

To avoid this problem, the data provider for the Tree component should point at the XMLDocumentObject's first child, as shown here:

```
myTree.dataProvider = myXML.firstChild;
```

Tree parameters

You can set the following authoring parameters for each Tree component instance in the Property inspector or in the Component inspector:

multipleSelection is a Boolean value that indicates whether a user can select multiple items (`true`) or not (`false`). The default value is `false`.

rowHeight indicates the height of each row, in pixels. The default value is 20.

You can write ActionScript to control these and additional options for the Tree component using its properties, methods, and events. For more information, see "Tree class (Flash Professional only)" on page 770.

You cannot enter data parameters in the Property inspector or in the Component inspector for the Tree component as you can with other components. For more information, see "Using the Tree component (Flash Professional only)" on page 759 and "Creating an application with the Tree component" on page 762.

Creating an application with the Tree component

The following procedures show how to use a Tree component to display mailboxes in an e-mail application.

The Tree component does not allow you to enter data parameters in the Property inspector or Component inspector. Because of the complexity of a Tree component's data structure, you must either import an XML object at runtime or build one in Flash while authoring. To create XML in Flash, you can use the TreeDataProvider interface, use the ActionScript XML object, or build an XML string. Each of these options is explained in the following procedures.

To add a Tree component to an application and load XML:

1. In Flash, select File > New and select Flash Document.
2. Save the document as **treeMenu.fla**.
3. In the Components panel, double-click the Tree component to add it to the Stage.
4. Select the Tree instance. In the Property inspector, enter the instance name **menuTree**.
5. Select the Tree instance and press F8. Select Movie Clip, and enter the name **TreeNavMenu**.
6. Click the Advanced button, and select Export for ActionScript.
7. Type **TreeNavMenu** in the AS 2.0 Class text box and click OK.
8. Select File > New and select ActionScript File.
9. Save the file as **TreeNavMenu.as** in the same directory as treeMenu.fla.
10. In the Script window, enter the following code:

```
import mx.controls.Tree;

class TreeNavMenu extends MovieClip {
  var menuXML:XML;
  var menuTree:Tree;
  function TreeNavMenu() {
    // Set up the appearance of the tree and event handlers
    menuTree.setStyle("fontFamily", "_sans");
    menuTree.setStyle("fontSize", 12);
    // Load the menu XML
    var treeNavMenu = this;
    menuXML = new XML();
    menuXML.ignoreWhite = true;
    menuXML.load("TreeNavMenu.xml");
    menuXML.onLoad = function() {
      treeNavMenu.onMenuLoaded();
    };
  }
  function change(event:Object) {
    if (menuTree == event.target) {
```

```
        var node = menuTree.selectedItem;
        // If this is a branch, expand/collapse it
        if (menuTree.getIsBranch(node)) {
          menuTree.setIsOpen(node, !menuTree.getIsOpen(node), true);
        }
        // If this is a hyperlink, jump to it
        var url = node.attributes.url;
        if (url) {
          getURL(url, "_top");
        }
        // Clear any selection
        menuTree.selectedNode = null;
      }
    }
    function onMenuLoaded() {
      menuTree.dataProvider = menuXML.firstChild;
      menuTree.addEventListener("change", this);
    }
}
```

This ActionScript sets up styles for the tree. An XML object is created to load the XML file that creates the tree's nodes. Then the `onLoad` event handler is defined to set the data provider to the contents of the XML file.

11. Create a new file called TreeNavMenu.xml in a text editor.
12. Enter the following code in the file:

```
<node>
  <node label="My Bookmarks">
    <node label="Macromedia Web site" url="http://www.macromedia.com" />
    <node label="MXNA blog aggregator" url="http://www.markme.com/mxna" />
  </node>
  <node label="Google" url="http://www.google.com" />
</node>
```

13. Save your documents and return to treeMenu.fla. Select Control > Test Movie to test the application.

To load XML from an external file:

1. In Flash, select File > New and select Flash Document.
2. Drag an instance of the Tree component onto the Stage.
3. Select the Tree instance. In the Property inspector, enter the instance name **myTree**.
4. In the Actions panel on Frame 1, enter the following code:

```
var myTreeDP:XML = new XML();
myTreeDP.ignoreWhite = true;
myTreeDP.load("treeXML.xml");
myTreeDP.onLoad = function() {
  myTree.dataProvider = this.firstChild;
};
var treeListener:Object = new Object();
treeListener.change = function(evt:Object) {
  trace("selected node is: "+evt.target.selectedNode);
  trace("");
```

```
};
myTree.addEventListener("change", treeListener);
```

This code creates an XML object called `myTreeDP` and calls the `XML.load()` method to load an XML data source. The code then defines an `onLoad` event handler that sets the `dataProvider` property of the `myTree` instance to the new XML object when the XML loads. For more information about the XML object, see its entry in *Flash ActionScript Language Reference*.

5. Create a new file called **treeXML.xml** in a text editor.
6. Enter the following code in the file:

```
<node>
  <node label="Mail">
    <node label="INBOX"/>
    <node label="Personal Folder">
      <node label="Business" isBranch="true" />
      <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>
```

7. Select Control > Test Movie.

In the SWF file, you can see the XML structure displayed in the tree. Click items in the tree to see the `trace()` statements in the `change` event handler send the data values to the Output panel.

To use the TreeDataProvider class to create XML in Flash while authoring:

1. In Flash, select File > New and select Flash Document.
2. Drag an instance of the Tree component onto the Stage.
3. Select the Tree instance and in the Property inspector, enter the instance name **myTree**.
4. In the Actions panel on Frame 1, enter the following code:

```
var myTreeDP:XML = new XML();
myTreeDP.addTreeNode("Local Folders", 0);
// Use XML.firstChild to nest child nodes below Local Folders
var myTreeNode:XMLNode = myTreeDP.firstChild;
myTreeNode.addTreeNode("Inbox", 1);
myTreeNode.addTreeNode("Outbox", 2);
myTreeNode.addTreeNode("Sent Items", 3);
myTreeNode.addTreeNode("Deleted Items", 4);
// Assign the myTreeDP data source to the myTree component
myTree.dataProvider = myTreeDP;
// Set each of the four child nodes to be branches
for (var i = 0; i<myTreeNode.childNodes.length; i++) {
  var node:XMLNode = myTreeNode.getTreeNodeAt(i);
  myTree.setIsBranch(node, true);
}
```

This code creates an XML object called `myTreeDP`. Any XML object on the same frame as a Tree component automatically receives all the properties and methods of the TreeDataProvider interface. The second line of code creates a single root node called Local Folders. For detailed information about the rest of the code, see the comments (lines preceded with `//`) throughout the code.

5. Select Control > Test Movie.

   In the SWF file, you can see the XML structure displayed in the tree. Click items in the tree to see the `trace()` statements in the `change` event handler send the data values to the Output panel.

To use the ActionScript XML class to create XML:

1. In Flash, select File > New and select Flash Document.
2. Drag an instance of the Tree component onto the Stage.
3. Select the Tree instance. In the Property inspector, enter the instance name **myTree**.
4. In the Actions panel on Frame 1, enter the following code:

```
// Create an XML object
var myTreeDP:XML = new XML();
// Create node values
var myNode0:XMLNode = myTreeDP.createElement("node");
myNode0.attributes.label = "Local Folders";
myNode0.attributes.data = 0;
var myNode1:XMLNode = myTreeDP.createElement("node");
myNode1.attributes.label = "Inbox";
myNode1.attributes.data = 1;
var myNode2:XMLNode = myTreeDP.createElement("node");
myNode2.attributes.label = "Outbox";
myNode2.attributes.data = 2;
var myNode3:XMLNode = myTreeDP.createElement("node");
myNode3.attributes.label = "Sent Items";
myNode3.attributes.data = 3;
var myNode4:XMLNode = myTreeDP.createElement("node");
myNode4.attributes.label = "Deleted Items";
myNode4.attributes.data = 4;
// Assign nodes to the hierarchy in the XML tree
myTreeDP.appendChild(myNode0);
myTreeDP.firstChild.appendChild(myNode1);
myTreeDP.firstChild.appendChild(myNode2);
myTreeDP.firstChild.appendChild(myNode3);
myTreeDP.firstChild.appendChild(myNode4);
// Assign the myTreeDP data source to the Tree component
myTree.dataProvider = myTreeDP;
```

   For more information about the XML object, see its entry in *Flash ActionScript Language Reference*.

5. Select Control > Test Movie.

   In the SWF file, you can see the XML structure displayed in the tree. Click items in the tree to see the `trace()` statements in the `change` event handler send the data values to the Output panel.

To use a well-formed string to create XML in Flash while authoring:

1. In Flash, select File > New and select Flash Document.
2. Drag an instance of the Tree component onto the Stage.
3. Select the Tree instance. In the Property inspector, enter the instance name **myTree**.
4. In the Actions panel on Frame 1, enter the following code:

```
var myTreeDP:XML = new XML("<node label='Local Folders'><node label='Inbox'
  data='0'/><node label='Outbox' data='1'/></node>");
myTree.dataProvider = myTreeDP;
```

   This code creates the XML object `myTreeDP` and assigns it to the `dataProvider` property of `myTree`.

5. Select Control > Test Movie.

   In the SWF file, you can see the XML structure displayed in the tree. Click items in the tree to see the `trace()` statements in the `change` event handler send the data values to the Output panel.

Customizing the Tree component (Flash Professional only)

You can transform a Tree component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`). When a tree isn't wide enough to display the text of the nodes, the text is clipped.

Using styles with the Tree component

A Tree component uses the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`backgroundColor`	Both	The background color of the list. The default color is white and is defined on the class style declaration. This style is ignored if `alternatingRowColors` is specified.
`backgroundDisabledColor`	Both	The background color when the component's `enabled` property is set to `"false"`. The default value is 0xDDDDDD (medium gray).
`depthColors`	Both	Sets the background colors for rows based on the depth of each node. The value is an array of colors where the first element is the background color for the root node, the second element is the background color for its children, and so on, continuing through the number of colors provided in the array. This style property is not set by default.

Style	Theme	Description
border styles	Both	The Tree component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The default border style is `"inset"`.
`color`	Both	The text color.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.
`fontWeight`	Both	The font weight: either `"none"` or `"bold"`. The default value is `"none"`. All components can also accept the value `"normal"` in place of `"none"` during a `setStyle()` call, but subsequent calls to `getStyle()` will return `"none"`.
`textAlign`	Both	The text alignment: either `"left"`, `"right"`, or `"center"`. The default value is `"left"`.
`textDecoration`	Both	The text decoration: either `"none"` or `"underline"`. The default value is `"none"`.
`textIndent`	Both	A number indicating the text indent. The default value is 0.
`defaultLeafIcon`	Both	The icon displayed in a Tree control for leaf nodes when no icon is specified for a particular node. The default value is `"TreeNodeIcon"`, which is an image representing a piece of paper.
`disclosureClosedIcon`	Both	The icon displayed next to a closed folder node in a Tree component. The default value is `"TreeDisclosureClosed"`, which is a gray arrow pointing to the right.
`disclosureOpenIcon`	Both	The icon displayed next to an open folder node in a Tree component. The default value is `"TreeDisclosureOpen"`, which is a gray arrow pointing down.
`folderClosedIcon`	Both	The icon displayed for a closed folder node in a Tree component if no node-specific icon is set. The default value is `"TreeFolderClosed"`, which is a yellow closed file folder image.

Style	Theme	Description
folderOpenIcon	Both	The icon displayed for an open folder node in a Tree component if no node-specific icon is set. The default value is "TreeFolderOpen", which is a yellow open file folder image.
indentation	Both	The number of pixels to indent each row of a Tree component. The default value is 17.
openDuration	Both	The duration, in milliseconds, of the expand and collapse animations. The default value is 250.
openEasing	Both	A reference to a tweening function that controls the expand and collapse animations. Defaults to sine in/out. For more information, see "Customizing component animations" on page 75.
repeatDelay	Both	The number of milliseconds of delay between when a user first presses a button in the scrollbar and when the action begins to repeat. The default value is 500 (half a second).
repeatInterval	Both	The number of milliseconds between automatic clicks when a user holds the mouse button down on a button in the scrollbar. The default value is 35.
rollOverColor	Both	The background color of a rolled-over row. The default value is 0xE3FFD6 (bright green) with the Halo theme and 0xAAAAAA (light gray) with the Sample theme. When themeColor is changed through a setStyle() call, the framework sets rollOverColor to a value related to the themeColor chosen.
selectionColor	Both	The background color of a selected row. The default value is a 0xCDFFC1 (light green) with the Halo theme and 0xEEEEEE (very light gray) with the Sample theme. When themeColor is changed through a setStyle() call, the framework sets selectionColor to a value related to the themeColor chosen.
selectionDuration	Both	The length of the transition from a normal state to a selected state or back from selected to normal, in milliseconds. The default value is 200.
selectionDisabledColor	Both	The background color of a selected row. The default value is a 0xDDDDDD (medium gray). Because the default value for this property is the same as the default for backgroundDisabledColor, the selection is not visible when the component is disabled unless one of these style properties is changed.
selectionEasing	Both	A reference to the easing equation used to control the transition between selection states. Applies only for the transition from a normal to a selected state. The default equation uses a sine in/out formula. For more information, see "Customizing component animations" on page 75.

Style	Theme	Description
textRollOverColor	Both	The color of text when the pointer rolls over it. The default value is 0x2B333C (dark gray). This style is important when you set rollOverColor, because the two must complement each other so that text is easily viewable during a rollover.
textSelectedColor	Both	The color of text in the selected row. The default value is 0x005F33 (dark gray). This style is important when you set selectionColor, because the two must complement each other so that text is easily viewable while selected.
useRollOver	Both	Determines whether rolling over a row activates highlighting. The default value is true.

Setting styles for all Tree components in a document

The Tree class inherits from the List class, which inherits from the ScrollSelectList class. The default class-level style properties are defined on the ScrollSelectList class, which the Menu component and all List-based components extend. You can set new default style values on this class directly, and the new settings will be reflected in all affected components.

```
_global.styles.ScrollSelectList.setStyle("backgroundColor", 0xFF00AA);
```

To set a style property on the Tree components only, you can create a new CSSStyleDeclaration instance and store it in _global.styles.DataGrid.

```
import mx.styles.CSSStyleDeclaration;
if (_global.styles.Tree == undefined) {
  _global.styles.Tree = new CSSStyleDeclaration();
}
_global.styles.Tree.setStyle("backgroundColor", 0xFF00AA);
```

When creating a new class-level style declaration, you will lose all default values provided by the ScrollSelectList declaration. This includes backgroundColor, which is required for supporting mouse events. To create a class-level style declaration and preserve defaults, use a for..in loop to copy the old settings to the new declaration.

```
var source = _global.styles.ScrollSelectList;
var target = _global.styles.Tree;
for (var style in source) {
  target.setStyle(style, source.getStyle(style));
}
```

For more information about class-level styles, see "Setting styles for a component class" on page 71.

Using skins with the Tree component

The Tree component uses an instance of RectBorder for its border and scroll bars for scrolling images. For more information about skinning these visual elements, see "RectBorder class" on page 647 and "Using skins with the UIScrollBar component" on page 831.

Tree class (Flash Professional only)

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > ScrollSelectList > List component > Tree

ActionScript Class Name mx.controls.Tree

The methods, properties, and events of the Tree class allow you to manage and manipulate Tree objects.

Method summary for the Tree class

The following table lists methods of the Tree class.

Method	Description
`Tree.addTreeNode()`	Adds a node to a Tree instance.
`Tree.addTreeNodeAt()`	Adds a node at a specific location in a Tree instance.
`Tree.getDisplayIndex()`	Returns the display index of a given node.
`Tree.getIsBranch()`	Specifies whether the folder is a branch (has a folder icon and an expander arrow).
`Tree.getIsOpen()`	Indicates whether a node is open or closed.
`Tree.getNodeDisplayedAt()`	Maps a display index of the tree onto the node that is displayed there.
`Tree.getTreeNodeAt()`	Returns a node on the root of the tree.
`Tree.refresh()`	Updates the tree.
`Tree.removeAll()`	Removes all nodes from a Tree instance and refreshes the tree.
`Tree.removeTreeNodeAt()`	Removes a node at a specified position and refreshes the tree.
`Tree.setIcon()`	Specifies an icon for the specified node.
`Tree.setIsBranch()`	Specifies whether a node is a branch (has a folder icon and expander arrow).
`Tree.setIsOpen()`	Opens or closes a node.

Methods inherited from the UIObject class

The following table lists the methods the Tree class inherits from the UIObject class. When calling these methods from the Tree object, use the form *`TreeInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.

Method	Description
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Tree class inherits from the UIComponent class. When calling these methods from the Tree object, use the form *TreeInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Methods inherited from the List class

The following table lists the methods the Tree class inherits from the List class. When calling these methods from the Tree object, use the form *TreeInstance.methodName*.

Method	Description
List.addItem()	Adds an item to the end of the list.
List.addItemAt()	Adds an item to the list at the specified index. With the Tree component, it is better to use Tree.addTreeNodeAt().
List.getItemAt()	Returns the item at the specified index.
List.removeAll()	Removes all items from the list.
List.removeItemAt()	Removes the item at the specified index.
List.replaceItemAt()	Replaces the item at the specified index with another item.
List.setPropertiesAt()	Applies the specified properties to the specified item.
List.sortItems()	Sorts the items in the list according to the specified compare function.
List.sortItemsBy()	Sorts the items in the list according to a specified property.

Property summary for the Tree class

The following table lists properties of the Tree class.

Property	Description
Tree.dataProvider	Specifies an XML data source.
Tree.firstVisibleNode	Specifies the first node at the top of the display.
Tree.selectedNode	Specifies the selected node in a Tree instance.
Tree.selectedNodes	Specifies the selected nodes in a Tree instance.

Properties inherited from the UIObject class

The following table lists the properties the Tree class inherits from the UIObject class. When accessing these properties from the Tree object, use the form *TreeInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Tree class inherits from the UIComponent class. When accessing these properties from the Tree object, use the form *TreeInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the List class

The following table lists the properties the Tree class inherits from the List class. When accessing these properties from the Tree object, use the form *TreeInstance.propertyName*.

Property	Description
List.cellRenderer	Assigns the class or symbol to use to display each row of the list.
List.dataProvider	The source of the list items.
List.hPosition	The horizontal position of the list.
List.hScrollPolicy	Indicates whether the horizontal scroll bar is displayed ("on") or not ("off").
List.iconField	A field in each item to be used to specify icons.
List.iconFunction	A function that determines which icon to use.
List.labelField	Specifies a field of each item to be used as label text.
List.labelFunction	A function that determines which fields of each item to use for the label text.
List.length	The number of items in the list. This property is read-only.
List.maxHPosition	The number of pixels the list can scroll to the right, when List.hScrollPolicy is set to "on".
List.multipleSelection	Indicates whether multiple selection is allowed in the list (true) or not (false).
List.rowCount	The number of rows that are at least partially visible in the list.
List.rowHeight	The pixel height of every row in the list.
List.selectable	Indicates whether the list is selectable (true) or not (false).
List.selectedIndex	The index of a selection in a single-selection list.
List.selectedIndices	An array of the selected items in a multiple-selection list.
List.selectedItem	The selected item in a single-selection list. This property is read-only.
List.selectedItems	The selected item objects in a multiple-selection list. This property is read-only.
List.vPosition	Scrolls the list so the topmost visible item is the number assigned.
List.vScrollPolicy	Indicates whether the vertical scroll bar is displayed ("on"), not displayed ("off"), or displayed when needed ("auto").

Event summary for the Tree class

The following table lists events of the Tree class.

Event	Description
Tree.nodeClose	Broadcast when a node is closed by a user.
Tree.nodeOpen	Broadcast when a node is opened by a user.

Events inherited from the UIObject class

The following table lists the events the Tree class inherits from the UIObject class.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Tree class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

Events inherited from the List class

The following table lists the events the Tree class inherits from the List class.

Event	Description
`List.change`	Broadcast whenever user interaction causes the selection to change.
`List.itemRollOut`	Broadcast when the pointer rolls over and then off of list items.
`List.itemRollOver`	Broadcast when the pointer rolls over list items.
`List.scroll`	Broadcast when a list is scrolled.

Tree.addTreeNode()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myTree.addTreeNode(label [, data])
```

Usage 2:

```
myTree.addTreeNode(child)
```

Parameters

label A string that displays the node, or an object with a label field (or whatever label field name is specified by the `labelField` property).

data An object of any type that is associated with the node. This parameter is optional.

child Any XMLNode object.

Returns

The added XML node.

Description

Method; adds a child node to the tree. The node is constructed either from the information supplied in the *label* and *data* parameters (Usage 1), or from the prebuilt child node, which is an XMLNode object (Usage 2). Adding a preexisting node removes the node from its previous location.

Calling this method refreshes the view.

Example

The following code adds a new node to the root of `myTree`. The second line of code moves a node from the root of `mySecondTree` to the root of `myTree`.

```
myTree.addTreeNode("Inbox", 3);
myTree.addTreeNode(mySecondTree.getTreeNodeAt(3));
```

Tree.addTreeNodeAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myTree.addTreeNodeAt(index, label [, data])
```

Usage 2:

```
myTree.addTreeNodeAt(index, child)
```

Parameters

index The zero-based index position (among the child nodes) at which the node should be added.

label A string that displays the node.

data An object of any type that is associated with the node. This parameter is optional.

child Any XMLNode object.

Returns

The added XML node.

Description

Method; adds a node at the specified location in the tree. The node is constructed either from the information supplied in the *label* and *data* parameters (Usage 1), or from the prebuilt XMLNode object (Usage 2). Adding a preexisting node removes the node from its previous location.

Calling this method refreshes the view.

Example

The following example adds a new node as the second child of the root of `myTree`. The second line moves a node from `mySecondTree` to become the fourth child of the root of `myTree`:

```
myTree.addTreeNodeAt(1, "Inbox", 3);
myTree.addTreeNodeAt(3,mySecondTree.getTreeNodeAt(3));
```

Tree.dataProvider

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.dataProvider
```

Description

Property; either XML or a string. If `dataProvider` is an XML object, it is added directly to the tree. If `dataProvider` is a string, it must contain valid XML that is read by the tree and converted to an XML object.

You can either load XML from an external source at runtime or create it in Flash while authoring. To create XML, you can use either the TreeDataProvider methods, or the built-in ActionScript XML class methods and properties. You can also create a string that contains XML.

XML objects that are on the same frame as a Tree component automatically contain the TreeDataProvider methods and properties. You can use the ActionScript XML or XMLNode object.

Example

The following example imports an XML file and assigns it to the `myTree` instance of the Tree component:

```
myTreeDP = new XML();
myTreeDP.ignoreWhite = true;
myTreeDP.load("http://myServer.myDomain.com/source.xml");
myTreeDP.onLoad = function(){
   myTree.dataProvider = myTreeDP;
}
```

Note: Most XML files contain white space. To make Flash ignore white space, you must set the `XML.ignoreWhite` property to `true`.

Tree.firstVisibleNode

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.firstVisibleNode = someNode
```

Description

Property; indicates the first node that is visible at the top of the tree display. Use this property to scroll the tree display to a desired position. If the specified node *`someNode`* is within a node that hasn't been expanded, setting `firstVisibleNode` has no effect. The default value is the first visible node or `undefined` if there is no visible node. The value of this property is an XMLNode object.

This property is an analogue to the `List.vPosition` property.

Example

The following example sets the scroll position to the top of the display:

```
myTree.firstVisibleNode = myTree.getTreeNodeAt(0);
```

Tree.getDisplayIndex()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.getDisplayIndex(node)
```

Parameters

node An XMLNode object.

Returns

The index of the specified node, or `undefined` if the node is not currently displayed.

Description

Method; returns the display index of the node specified in the *node* parameter.

The display index indicates the item's position in the list of items that are visible in the tree window. For example, any children of a closed node are not in the display index. The list of display indices starts with 0 and proceeds through the visible items regardless of parent. In other words, the index is the row number, starting with 0, of the displayed rows.

Example

The following code gets the display index of `myNode`:

```
var x = myTree.getDisplayIndex(myNode);
```

Tree.getIsBranch()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.getIsBranch(node)
```

Parameters

node An XMLNode object.

Returns

A Boolean value that indicates whether the node is a branch (`true`) or not (`false`).

Description

Method; indicates whether the specified node has a folder icon and expander arrow (and is therefore a branch). This is set automatically when children are added to the node. You only need to call `Tree.setIsBranch()` to create empty folders.

Example

The following code assigns the node state to a variable:

```
var open = myTree.getIsBranch(myTree.getTreeNodeAt(1));
```

See also

`Tree.setIsBranch()`

Tree.getIsOpen()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.getIsOpen(node)
```

Parameters

node An XMLNode object.

Returns

A Boolean value that indicates whether the tree is open (`true`) or closed (`false`).

Description

Method; indicates whether the specified node is open or closed.

Note: Display indices change every time nodes open and close.

Example

The following code assigns the state of the node to a variable:

```
var open = myTree.getIsOpen(myTree.getTreeNodeAt(1));
```

Tree.getNodeDisplayedAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.getNodeDisplayedAt(index)
```

Parameters

index An integer representing the display position in the viewable area of the tree. This number is zero-based; the node at the first position is 0, second position is 1, and so on.

Returns

The specified XMLNode object.

Description

Method; maps a display index of the tree onto the node that is displayed there. For example, if the fifth row of the tree showed a node that is eight levels deep into the hierarchy, that node would be returned by a call to `getNodeDisplayedAt(4)`.

The display index is an array of items that can be viewed in the tree window. For example, any children of a closed node are not in the display index. The display index starts with 0 and proceeds through the visible items regardless of parent. In other words, the display index is the row number, starting with 0, of the displayed rows.

Note: Display indices change every time nodes open and close.

Example

The following code gets a reference to the XML node that is the second row displayed in `myTree`:

```
myTree.getNodeDisplayedAt(1);
```

Tree.getTreeNodeAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.getTreeNodeAt(index)
```

Parameters

index The index number of a node.

Returns

An XMLNode object.

Description

Method; returns the specified node on the root of `myTree`.

Example

The following code gets the second node on the first level in the tree `myTree`:

```
myTree.getTreeNodeAt(1);
```

Tree.nodeClose

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.nodeClose = function(eventObject){
  // insert your code here
```

```
}
myTreeInstance.addEventListener("nodeClose", listenerObject)
```

Description

Event; broadcast to all registered listeners when the nodes of a Tree component are closed by a user.

Version 2 components use a dispatcher/listener event model. The Tree component broadcasts a `nodeClose` event when one of its nodes is clicked closed; the event is handled by a function, also called a *handler*, that is attached to a listener object (*`listenerObject`*) that you create.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Tree.nodeClose` event's event object has one additional property: `node` (the XML node that closed).

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myTreeListener` is defined and passed to the `myTree.addEventListener()` method as the second parameter. The event object is captured by the `nodeClose` handler in the `evtObject` parameter. When the `nodeClose` event is broadcast, a `trace` statement is sent to the Output panel.

```
myTreeListener = new Object();
myTreeListener.nodeClose = function(evtObject){
  trace(evtObject.node + " node was closed");
}
myTree.addEventListener("nodeClose", myTreeListener);
```

Tree.nodeOpen

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.nodeOpen = function(eventObject){
  // insert your code here
}
myTreeInstance.addEventListener("nodeOpen", listenerObject)
```

Description

Event; broadcast to all registered listeners when a user opens a node on a Tree component.

Version 2 components use a dispatcher/listener event model. The Tree component dispatches a `nodeOpen` event when a node is clicked open by a user; the event is handled by a function, also called a *handler*, that is attached to a listener object (*`listenerObject`*) that you create. You call the `addEventListener()` method and pass it the name of the handler as a parameter.

When the event is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. The `Tree.nodeOpen` event's event object has one additional property: `node` (the XML node that was opened).

For more information, see "EventDispatcher class" on page 415.

Example

In the following example, a handler called `myTreeListener` is defined and passed to `myTree.addEventListener()` as the second parameter. The event object is captured by the `nodeOpen` handler in the `evtObject` parameter. When the `nodeOpen` event is broadcast, a `trace` statement is sent to the Output panel.

```
myTreeListener = new Object();
myTreeListener.nodeOpen = function(evtObject){
  trace(evtObject.node + " node was opened");
}
myTree.addEventListener("nodeOpen", myTreeListener);
```

Tree.refresh()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.refresh()
```

Parameters

None.

Returns

Nothing.

Description

Method; updates the tree.

Tree.removeAll()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.removeAll()
```

Parameters

None.

Returns

Nothing.

Description

Method; removes all nodes and refreshes the tree.

Example

The following code empties `myTree`:

```
myTree.removeAll();
```

Tree.removeTreeNodeAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.removeTreeNodeAt(index)
```

Parameters

index The index number of a tree child. Each child of a tree is assigned a zero-based index in the order in which it was created.

Returns

An XMLNode object, or `undefined` if an error occurs.

Description

Method; removes a node (specified by its index position) on the root of the tree and refreshes the tree.

Example

The following code removes the fourth child of the root of the tree `myTree`:

```
myTree.removeTreeNodeAt(3);
```

Tree.selectedNode

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.selectedNode
```

Description

Property; specifies the selected node in a tree instance.

Example

The following example specifies the first child node in `myTree`:

```
myTree.selectedNode = myTree.getTreeNodeAt(0);
```

See also

`Tree.selectedNodes`

Tree.selectedNodes

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.selectedNodes
```

Description

Property; specifies the selected nodes in a tree instance.

Example

The following example selects the first and third child nodes in `myTree`:

```
myTree.selectedNodes = [myTree.getTreeNodeAt(0), myTree.getTreeNodeAt(2)];
```

See also

`Tree.selectedNode`

Tree.setIcon()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.setIcon(node, linkID [, linkID2])
```

Parameters

node An XML node.

linkID The linkage identifier of a symbol to be used as an icon beside the node. This parameter is used for leaf nodes and for the closed state of branch nodes.

linkID2 For a branch node, the linkage identifier of a symbol to be used as an icon that represents the open state of the node. This parameter is optional.

Returns

Nothing.

Description

Method; specifies an icon for the specified node. This method takes one ID parameter (*linkID*) for leaf nodes and two ID parameters (*linkID* and *linkID2*) for branch nodes (the closed and open icons). For leaf nodes, the second parameter is ignored. For branch nodes, if you omit *linkID2*, the icon is used for both the closed and open states.

Example

The following code specifies that a symbol with the linkage identifier `imageIcon` be used beside the second node of `myTree`:

```
myTree.setIcon(myTree.getTreeNodeAt(1), "imageIcon");
```

Tree.setIsBranch()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.setIsBranch(node, isBranch)
```

Parameters

node An XML node.

isBranch A Boolean value indicating whether the node is (`true`) or is not (`false`) a branch.

Returns

Nothing.

Description

Method; specifies whether the node has a folder icon and expander arrow and either has children or can have children. A node is automatically set as a branch when it has children; you only need to call `setIsBranch()` when you want to create an empty folder. You may want to create branches that don't yet have children if, for example, you only want child nodes to load when a user opens a folder.

Calling `setIsBranch()` refreshes any views.

Example

The following code makes a node of `myTree` a branch:

```
myTree.setIsBranch(myTree.getTreeNodeAt(1), true);
```

Tree.setIsOpen()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myTree.setIsOpen(node, open [, animate])
```

Parameters

`node` An XML node.

`open` A Boolean value that opens a node (`true`) or closes it (`false`).

`animate` A Boolean value that determines whether the opening transition is animated (`true`) or not (`false`). This parameter is optional.

Returns

Nothing.

Description

Method; opens or closes a node.

Example

The following code opens a node of `myTree`:

```
myTree.setIsOpen(myTree.getTreeNodeAt(1), true);
```

TreeDataProvider interface (Flash Professional only)

The TreeDataProvider interface is a set of properties and methods and does not need to be instantiated to be used. If a Tree class is packaged in a SWF file, all XML instances in the SWF file contain the TreeDataProvider interface. All nodes in a tree are XML objects that contain the TreeDataProvider interface.

It's best to use the TreeDataProvider methods to create XML for the `Tree.dataProvider` property, because only TreeDataProvider broadcasts events that refresh the tree's display. These are events that the Tree class handles; you do not need to write functions to handle these events. (The built-in XML class methods don't broadcast such events.)

Use the TreeDataProvider methods to control the data model and the data display. Use built-in XML class methods for read-only tasks such as traversing through the tree hierarchy.

You can select the property that holds the text to be displayed by specifying a `labelField` or `labelFunction` property. For example, the code `myTree.labelField = "firstName";` results in the value of the property `myTreeDP.attributes.fred` being queried for the display text.

Method summary for the TreeDataProvider interface

The following table lists the methods of the TreeDataProvider interface.

Method	Description
`TreeDataProvider.addTreeNode()`	Adds a child node at the root of the tree.
`TreeDataProvider.addTreeNodeAt()`	Adds a child node at a specified location on the parent node.
`TreeDataProvider.getTreeNodeAt()`	Returns the specified child of a node.
`TreeDataProvider.removeTreeNode()`	Removes a node and all the node's descendants from the node's parent.
`TreeDataProvider.removeTreeNodeAt()`	Removes a node and all the node's descendants from the index position of the child node.

Property summary for the TreeDataProvider interface

The following table lists the properties of the TreeDataProvider interface.

Property	Description
`TreeDataProvider.attributes.data`	Specifies the data to associate with a node.
`TreeDataProvider.attributes.label`	Specifies the text to be displayed next to a node.

TreeDataProvider.addTreeNode()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
someNode.addTreeNode(label, data)
```

Usage 2:

```
someNode.addTreeNode(child)
```

Parameters

label A string that displays the node.

data An object of any type that is associated with the node.

child Any XMLNode object.

Returns

The added XML node.

Description

Method; adds a child node at the root of the tree. The node is constructed either from the information supplied in the *label* and *data* parameters (Usage 1), or from the prebuilt child node, which is an XMLNode object (Usage 2). Adding a preexisting node removes the node from its previous location.

Calling this method refreshes the display of the tree.

Example

The first line of code in the following example locates the node to which to add a child. The second line adds a new node to a specified node.

```
var myTreeNode = myTreeDP.firstChild.firstChild;
myTreeNode.addTreeNode("Inbox", 3);
```

The following code moves a node from one tree to the root of another tree:

```
myTreeNode.addTreeNode(mySecondTree.getTreeNodeAt(3));
```

TreeDataProvider.addTreeNodeAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
someNode.addTreeNodeAt(index, label, data)
```

Usage 2:

```
someNode.addTreeNodeAt(index, child)
```

Parameters

index An integer that indicates the index position (among the child nodes) at which the node should be added.

label A string that displays the node.

data An object of any type that is associated with the node.

child Any XMLNode object.

Returns

The added XML node.

Description

Method; adds a child node at the specified location in the parent node. The node is constructed either from the information supplied in the *label* and *data* parameters (Usage 1), or from the prebuilt child node, which is an XMLNode object (Usage 2). Adding a preexisting node removes the node from its previous location.

Calling this method refreshes the display of the tree.

Example

The following code locates the node to which you will add a node and adds a new node as the second child of the root:

```
var myTreeNode = myTreeDP.firstChild.firstChild;
myTreeNode.addTreeNodeAt(1, "Inbox", 3);
```

The following code moves a node from one tree to become the fourth child of the root of another tree:

```
myTreeNode.addTreeNodeAt(3, mySecondTree.getTreeNodeAt(3));
```

TreeDataProvider.attributes.data

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
someNode.attributes.data
```

Description

Property; specifies the data to associate with the node. This adds the value as an attribute in the XMLNode object. Setting this property does not refresh any tree displays. This property can be of any data type.

Example

The following code locates the node to adjust and sets its `data` property:

```
var myTreeNode = myTreeDP.firstChild.firstChild;
myTreeNode.attributes.data = "hi"; // results in <node data = "hi">;
```

See also

`TreeDataProvider.attributes.label`

TreeDataProvider.attributes.label

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
someNode.attributes.label
```

Description

Property; a string that specifies the text displayed for the node. This is written to an attribute of the XMLNode object. Setting this property does not refresh any tree displays.

Example

The following code locates the node to adjust and sets its `label` property. The result of the following code is `<node label="Mail">`.

```
var myTreeNode = myTreeDP.firstChild.firstChild;
myTreeNode.attributes.label = "Mail";
```

See also

`TreeDataProvider.attributes.data`

TreeDataProvider.getTreeNodeAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
someNode.getTreeNodeAt(index)
```

Parameters

index An integer representing the position of the child node in the current node.

Returns

The specified node.

Description

Method; returns the specified child node of the node.

Example

The following code locates a node and then retrieves the second child of `myTreeNode`:

```
var myTreeNode = myTreeDP.firstChild.firstChild;
myTreeNode.getTreeNodeAt(1);
```

TreeDataProvider.removeTreeNode()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
someNode.removeTreeNode()
```

Parameters

None.

Returns

The removed XML node, or `undefined` if an error occurs.

Description

Method; removes the specified node, and any of its descendants, from the node's parent.

Example

The following code removes a node:

```
myTreeDP.firstChild.removeTreeNode();
```

TreeDataProvider.removeTreeNodeAt()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
someNode.removeTreeNodeAt(index)
```

Parameters

index An integer indicating the position of the node to be removed.

Returns

The removed XML node, or `undefined` if an error occurs.

Description

Method; removes a node (and all of its descendants) specified by the current node and index position of the child node. Calling this method refreshes the view.

Example

The following code removes the fourth child of the `myTreeDP.firstChild` node:

```
myTreeDP.firstChild.removeTreeNodeAt(3);
```

UIComponent class

The UIComponent class does not represent a visual component; it contains methods, properties, and events that allow Macromedia components to share some common behavior. All version 2 components extend UIComponent. The UIComponent class lets you do the following:

- Receive focus and keyboard input
- Enable and disable components
- Resize by layout

To use the methods and properties of UIComponent, you call them directly from whichever component you are using. For example, to call `UIComponent.setFocus()` from the RadioButton component, you would write the following code:

```
myRadioButton.setFocus();
```

You only need to create an instance of UIComponent if you are using version 2 of the Macromedia Component Architecture to create a new component. Even in that case, UIComponent is often created implicitly by other subclasses such as Button. If you do need to create an instance of UIComponent, use the following code:

```
class MyComponent extends mx.core.UIComponent;
```

UIComponent class (API)

Inheritance MovieClip > UIObject class > UIComponent

ActionScript Class Name mx.core.UIComponent

The methods, properties, and events of the UIComponent class allow you to control the common behavior of Flash visual components.

Method summary for the UIComponent class

The following table lists methods of the UIComponent class.

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Methods inherited from the UIObject class

The following table lists the methods the UIComponent class inherits from the UIObject class. When calling these methods from the UIComponent object, use the form *`UIComponentInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.

Method	Description
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Property summary for the UIComponent class

The following table lists properties of the UIComponent class.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Properties inherited from the UIObject class

The following table lists the properties the UIComponent class inherits from the UIObject class. When accessing these properties from the UIComponent object, use the form *UIComponentInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.

Property	Description
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Event summary for the UIComponent class

The following table lists events of the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Events inherited from the UIObject class

The following table lists the events the UIComponent class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

UIComponent.enabled

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.enabled

Description

Property; indicates whether the component can (`true`) or cannot (`false`) accept focus and mouse input. The default value is `true`.

Example

The following example sets the `enabled` property of a CheckBox component to `false`:

```
checkBoxInstance.enabled = false;
```

UIComponent.focusIn

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(focusIn){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.focusIn = function(eventObject){
  ...
}
componentInstance.addEventListener("focusIn", listenerObject)
```

Description

Event; notifies listeners that the object has received keyboard focus.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*`componentInstance`*) dispatches an event (in this case, `focusIn`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following code disables a Button component, `btn`, while a user types in the TextInput component, `txt`, and enables the button when the user click on it:

```
var txt:mx.controls.TextInput;
var btn:mx.controls.Button;

var txtListener:Object = new Object();
txtListener.focusOut = function() {
  _root.btn.enabled = true;
}
txt.addEventListener("focusOut", txtListener);

var txtListener2:Object = new Object();
txtListener2.focusIn = function() {
  _root.btn.enabled = false;
}
txt.addEventListener("focusIn", txtListener2);
```

See also

`EventDispatcher.addEventListener()`, `UIComponent.focusOut`

UIComponent.focusOut

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
on(focusOut){
  ...
}
listenerObject = new Object();
listenerObject.focusOut = function(eventObject){
  ...
}
componentInstance.addEventListener("focusOut", listenerObject)
```

Description

Event; notifies listeners that the object has lost keyboard focus.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*componentInstance*) dispatches an event (in this case, `focusOut`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following code disables a Button component, `btn`, while a user types in the TextInput component, `txt`, and enables the button when the user click on it:

```
var txt:mx.controls.TextInput;
var btn:mx.controls.Button;

var txtListener:Object = new Object();
txtListener.focusOut = function() {
  _root.btn.enabled = true;
}
txt.addEventListener("focusOut", txtListener);

var txtListener2:Object = new Object();
txtListener2.focusIn = function() {
  _root.btn.enabled = false;
}
txt.addEventListener("focusIn", txtListener2);
```

See also

`EventDispatcher.addEventListener()`, `UIComponent.focusIn`

UIComponent.getFocus()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.getFocus();
```

Parameters

None.

Returns

A reference to the object that currently has focus.

Description

Method; returns a reference to the object that has keyboard focus.

Example

The following code returns a reference to the object that has focus and assigns it to the `tmp` variable:

```
var tmp = checkbox.getFocus();
```

UIComponent.keyDown

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
on(keyDown){
  ...
}
listenerObject = new Object();
listenerObject.keyDown = function(eventObject){
  ...
}
componentInstance.addEventListener("keyDown", listenerObject)
```

Description

Event; notifies listeners when a key is pressed. This is a very low-level event that you should not use unless necessary, because it can affect system performance.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*`componentInstance`*) dispatches an event (in this case, `keyDown`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following code makes an icon blink when a key is pressed:

```
formListener.handleEvent = function(eventObj)
{
  form.icon.visible = !form.icon.visible;
}
form.addEventListener("keyDown", formListener);
```

UIComponent.keyUp

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
on(keyUp){
  ...
}
listenerObject = new Object();
listenerObject.keyUp = function(eventObject){
  ...
}
componentInstance.addEventListener("keyUp", listenerObject)
```

Description

Event; notifies listeners when a key is released. This is a low-level event that you should not use unless necessary, because it can affect system performance.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*`componentInstance`*) dispatches an event (in this case, `keyUp`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following code makes an icon blink when a key is released:

```
formListener.handleEvent = function(eventObj)
```

```
{
  form.icon.visible = !form.icon.visible;
}
form.addEventListener("keyUp", formListener);
```

UIComponent.setFocus()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.setFocus();
```

Parameters

None.

Returns

Nothing.

Description

Method; sets the focus to this component instance. The instance with focus receives all keyboard input.

Example

The following code gives focus to the `checkbox` instance:

```
checkbox.setFocus();
```

UIComponent.tabIndex

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
instance.tabIndex
```

Description

Property; a number indicating the tabbing order for a component in a document.

Example

The following code sets the value of `tmp` to the `tabIndex` property of the `checkbox` instance:

```
var tmp = checkbox.tabIndex;
```

UIEventDispatcher class

ActionScript Class Name mx.events.UIEventDispatcher

Inheritance EventDispatcher class > UIEventDispatcher

The UIEventDispatcher class is mixed in to the UIComponent class and allows components to emit certain events.

If you want an object that doesn't inherit from UIComponent to dispatch certain events, you can use UIEventDispatcher.

Method summary for the UIEventDispatcher class

The following table lists the method of the UIEventDispatcher class.

Method	Description
`UIEventDispatcher.removeEventListener()`	Removes a registered listener from a component instance. This method overrides the `eventDispatcher.removeEventListenter()` method.

Methods inherited from the EventDispatcher class

The following table lists the methods the UIEventDispatcher class inherits from the EventDispatcher class. When calling these methods from the UIEventDispatcher object, use the form *`UIEventDispatcherInstance.methodName`*.

Method	Description
`EventDispatcher.addEventListener()`	Registers a listener to a component instance.
`EventDispatcher.dispatchEvent()`	Dispatches an event to all registered listeners.

Event summary for the UIEventDispatcher class

The following table lists events of the UIEventDispatcher class.

Method	Description
`UIEventDispatcher.keyDown`	Broadcast when a key is pressed.
`UIEventDispatcher.keyUp`	Broadcast when a pressed key is released.
`UIEventDispatcher.load`	Broadcast when a component loads into Flash Player.
`UIEventDispatcher.mouseDown`	Broadcast when the mouse is pressed.
`UIEventDispatcher.mouseOut`	Broadcast when the mouse is moved off a component instance.
`UIEventDispatcher.mouseOver`	Broadcast when the mouse is moved over a component instance.
`UIEventDispatcher.mouseUp`	Broadcast when the mouse is pressed and released.
`UIEventDispatcher.unload`	Broadcast when a component is unloaded from Flash Player.

UIEventDispatcher.keyDown

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.keyDown = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("keyDown", listenerObject)
```

Description

Event; broadcast to all registered listeners when a key is pressed and the Flash application has focus.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event.

UIEventDispatcher.keyUp

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.keyUp = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("keyUp", listenerObject)
```

Description

Event; broadcast to all registered listeners when a key that was pressed is released and the Flash application has focus.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event.

UIEventDispatcher.load

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.load = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("load", listenerObject)
```

Description

Event; broadcast to all registered listeners when a component is loaded into Flash Player.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event.

UIEventDispatcher.mouseDown

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.mouseDown = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("mouseDown", listenerObject)
```

Description

Event; broadcast to all registered listeners when a Flash application has focus and the mouse is pressed.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event.

UIEventDispatcher.mouseOut

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.mouseOut = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("mouseOut", listenerObject)
```

Description

Event; broadcast to all registered listeners when a Flash application has focus and the mouse is moved off a component instance.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event.

UIEventDispatcher.mouseOver

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.mouseOver = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("mouseOver", listenerObject)
```

Description

Event; broadcast to all registered listeners when a Flash application has focus and the mouse is moved over a component instance.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event.

UIEventDispatcher.mouseUp

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.mouseUp = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("mouseUp", listenerObject)
```

Description

Event; broadcast to all registered listeners when a Flash application has focus and the mouse is pressed and released.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. For more information, see "EventDispatcher class" on page 415.

UIEventDispatcher.removeEventListener()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004 and Flash MX Professional 2004.

Usage

```
componentInstance.removeEventListener(event, listener)
```

Parameters

event A string that is the name of the event.

listener A reference to a listener object or function.

Returns

Nothing.

Description

Method; unregisters a listener object from a component instance that is broadcasting an event. This method overrides the `EventDispatcher.removeEventListener()` event found in the EventDispatcher base class.

UIEventDispatcher.unload

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
listenerObject = new Object();
listenerObject.unload = function(eventObject){
  // insert your code here
}
componentInstance.addEventListener("unload", listenerObject)
```

Description

Event; broadcast to all registered listeners when a component is unloaded from Flash Player.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event.

UIObject class

Inheritance MovieClip > UIObject

ActionScript Class Name mx.core.UIObject

UIObject is the base class for all version 2 components; it is not a visual component. The UIObject class wraps the ActionScript MovieClip object and contains functions and properties that allow version 2 components to share some common behavior. Wrapping the MovieClip class allows Macromedia to add new events and extend functionality in the future without breaking content. Wrapping the MovieClip class also allows users who aren't familiar with the traditional Flash concepts of "movie" and "frame" to use properties, methods, and events to create component-based applications without learning those concepts.

The UIObject class implements the following:

- Styles
- Events
- Resize by scaling

To use the methods and properties of the UIObject class, you call them directly from whichever component you are using. For example, to call the `UIObject.setSize()` method from the RadioButton component, you would write the following code:

```
myRadioButton.setSize(30, 30);
```

You only need to create an instance of UIObject if you are using version 2 of the Macromedia Component Architecture to create a new component. Even in that case, UIObject is often created implicitly by other subclasses like Button. If you do need to create an instance of UIObject, use the following code:

```
class MyComponent extends UIObject;
```

Method summary for the UIObject class

The following table lists methods of the UIObject class.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.

Method	Description
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Property summary for the UIObject class

The following table lists properties of the UIObject class.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Event summary for the UIObject class

The following table lists events of the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

UIObject.bottom

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.bottom
```

Description

Property (read-only); a number indicating the bottom position of the object, in pixels, relative to its parent's bottom. To set this property, call `UIObject.move()`.

Example

This example moves the check box so it aligns under the bottom edge of the list box:

```
myCheckbox.move(myCheckbox.x, form.height - listbox.bottom);
```

UIObject.createClassObject()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.createClassObject(className, instanceName, depth,
  initObject)
```

Parameters

`className` An object indicating the class of the new instance.

`instanceName` A string indicating the instance name of the new instance.

`depth` A number indicating the depth of the new instance.

`initObject` An object containing initialization properties for the new instance.

Returns

A UIObject object that is an instance of the specified class.

Description

Method; creates an instance of a component at runtime. You need to use the `import` statement and specify the class package name before calling this method. In addition, the component must be in the FLA file's library.

Example

The following code imports the assets of the Button component and then makes a subobject of the Button component.

```
import mx.controls.Button;
createClassObject(Button,"button2",5,{label:"Test Button"});
```

The following example creates a CheckBox object:

```
import mx.controls.CheckBox;
form.createClassObject(CheckBox, "cb", 0, {label:"Check this"});
```

You can also specify the class package name using the following syntax:

```
createClassObject(mx.controls.Button,"button2",5,{label:"Test Button"});
```

UIObject.createObject()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.createObject(*linkageName*, *instanceName*, *depth*, *initObject*)

Parameters

linkageName A string indicating the linkage identifier of a symbol in the library.

instanceName A string indicating the instance name of the new instance.

depth A number indicating the depth of the new instance.

initObject An object containing initialization properties for the new instance.

Returns

A UIObject object that is an instance of the symbol.

Description

Method; creates a subobject on an object. This method is generally used only by component developers or advanced developers.

Example

The following example creates a CheckBox instance on the `form` object:

```
form.createObject("CheckBox", "sym1", 0);
```

UIObject.destroyObject()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.destroyObject(instanceName)
```

Parameters

instanceName A string indicating the instance name of the object to be destroyed.

Returns

Nothing.

Description

Method; destroys a component instance.

UIObject.doLater()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.doLater(target, "function")
```

Parameters

target A reference to a Timeline that contains the specified function.

function A string indicating a function name to be called after a frame has passed.

Returns

Nothing.

Description

Method; calls a user-defined function only after the component has finished setting all of its properties from the Property inspector or Component inspector. All version 2 components that inherit from UIObject have the `doLater()` method.

Component properties set in the Property inspector or Component inspector may not be immediately available to ActionScript in the Timeline. For example, attempting to trace the `label` property from a CheckBox component using ActionScript on the first frame of your SWF fails without notification, even though the component appears on the Stage as expected.

Although properties that are set in a class or a frame script are available immediately, most properties assigned in the Property inspector or Component inspector are not set until the next frame within the component itself.

Although any approach that delays access of the property will resolve this problem, the simplest and most direct solution is to use the `doLater()` method.

Example

The following example shows how the `doLater()` method is used:

```
// doLater() is called from the component instance

myCheckBox.doLater (this, "delay");

// the function or method called from doLater()

function delay() {
  trace(myCheckBox.label); // the property can now be traced
  // any additional statements go here
}
```

UIObject.draw

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
on(draw){
   ...
}
listenerObject = new Object();
listenerObject.draw = function(eventObject){
   ...
}
componentInstance.addEventListener("draw", listenerObject)
```

Description

Event; notifies listeners that the object is about to draw its graphics. This is a low-level event that you should not use unless necessary, because it can affect system performance.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*componentInstance*) dispatches an event (in this case, `draw`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following code redraws the object `form2` when the `form` object is drawn:

```
formListener.draw = function(eventObj){
  form2.redraw(true);
}
form.addEventListener("draw", formListener);
```

See also

`EventDispatcher.addEventListener()`

UIObject.getStyle()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance`.getStyle(`*propertyName*`)`

Parameters

propertyName A string indicating the name of the style property (for example, `"fontWeight"`, `"borderStyle"`, and so on).

Returns

The value of the style property. The value can be of any data type.

Description

Method; gets the style property from the style declaration or object. If the style property is an inheriting style, the ancestors of the object may be the source of the style value.

For a list of the styles supported by each component, see the individual component entries. See also "Using global, custom, and class styles in the same document" on page 73.

Example

The following code sets the `ib` instance's `fontWeight` style property to bold if the `cb` instance's `fontWeight` style property is bold:

```
if (cb.getStyle("fontWeight") == "bold")
{
   ib.setStyle("fontWeight", "bold");
};
```

UIObject.height

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.height

Description

Property (read-only); a number indicating the height of the object, in pixels. To change the `height` property, call `UIObject.setSize()`.

Example

The following example makes the check box taller:

```
myCheckbox.setSize(myCheckbox.width, myCheckbox.height + 10);
```

UIObject.hide

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
on(hide){
   ...
}
listenerObject = new Object();
listenerObject.hide = function(eventObject){
   ...
}
componentInstance.addEventListener("hide", listenerObject)
```

Description

Event; broadcast when the object's `visible` property is changed from `true` to `false`.

Example

The following handler displays a message in the Output panel when the object it's attached to becomes invisible.

```
on(hide) {
  trace("I've become invisible.");
}
```

See also

UIObject.reveal

UIObject.invalidate()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.invalidate()

Parameters

None.

Returns

Nothing.

Description

Method; marks the object so it will be redrawn on the next frame interval.

This method is primarily useful to developers of new custom components. A custom component is likely to support a number of operations that change the component's appearance.

Often, the best way to build a component is to centralize the logic for updating the component's appearance in the `draw()` method. If the component has a `draw()` method, you can call `invalidate()` on the component to redraw it. (For information on defining a `draw()` method, see "Defining core functions" on page 944.)

All operations that change the component's appearance can call `invalidate()` instead of redrawing the component themselves. This has some advantages: code isn't duplicated unnecessarily, and multiple changes can easily be batched up into one redraw, instead of causing multiple, redundant redraws.

Example

The following example marks the ProgressBar instance `pBar` for redrawing:

```
pBar.invalidate();
```

UIObject.left

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.left
```

Description

Property (read-only); a number indicating the left edge of the object, in pixels, relative to its parent. To set this property, call `UIObject.move()`.

UIObject.load

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(load){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.load = function(eventObject){
   ...
}
componentInstance.addEventListener("load", listenerObject)
```

Description

Event; notifies listeners that the subobject for this object is being created.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*componentInstance*) dispatches an event (in this case, `load`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example creates an instance of MySymbol once the `form` instance is loaded:

```
formListener.handleEvent = function(eventObj)
{
  form.createObject("MySymbol", "sym1", 0);
}
form.addEventListener("load", formListener);
```

UIObject.move

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(move){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.move = function(eventObject){
  ...
}
componentInstance.addEventListener("move", listenerObject)
```

Description

Event; notifies listeners that the object has moved.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*componentInstance*) dispatches an event (in this case, move) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the EventDispatcher.addEventListener() method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example calls the move() method to keep form2 100 pixels down and to the right of form1:

```
formListener.handleEvent = function(){
  form2.move(form1.x + 100, form1.y + 100);
}
form1.addEventListener("move", formListener);
```

UIObject.move()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.move(*x*, *y*)

Parameters

x A number that indicates the position of the object's upper left corner, relative to its parent.

y A number that indicates the position of the object's upper left corner, relative to its parent.

Returns

Nothing.

Description

Method; moves the object to the requested position. You should pass only integral values to UIObject.move(), or the component may appear fuzzy.

Example

This example move the check box 10 pixels to the right:

```
myCheckbox.move(myCheckbox.x + 10, myCheckbox.y);
```

UIObject.redraw()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.redraw(always)
```

Parameters

always A Boolean value. If `true`, the method draws the object, even if `invalidate()` wasn't called. If `false`, the method draws the object only if `invalidate()` was called.

Returns

Nothing.

Description

Method; forces validation of the object so that it is drawn in the current frame.

Example

The following example creates a check box and a button and draws them because other scripts are not expected to modify the form:

```
form.createClassObject(mx.controls.CheckBox, "cb", 0);
form.createClassObject(mx.controls.Button, "b", 1);
form.redraw(true)
```

UIObject.resize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(resize){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.resize = function(eventObject){
  ...
}
componentInstance.addEventListener("resize", listenerObject)
```

Description

Event; notifies listeners that an object has been resized.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*`componentInstance`*) dispatches an event (in this case, `resize`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example calls the `setSize()` method to make `sym1` half the width and a fourth of the height when `form` is moved:

```
formListener.handleEvent = function(eventObj){
  form.sym1.setSize(sym1.width / 2, sym1.height / 4);
}
form.addEventListener("resize", formListener);
```

UIObject.reveal

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
on(reveal){
  ...
}
listenerObject = new Object();
listenerObject.reveal = function(eventObject){
  ...
}
componentInstance.addEventListener("reveal", listenerObject)
```

Description

Event; broadcast when the object's `visible` property changes from `false` to `true`.

Example

The following handler displays a message in the Output panel when the object it's attached to becomes visible.

```
on(reveal) {
  trace("I've become visible.");
}
```

See also

UIObject.hide

UIObject.right

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.right

Description

Property (read-only); a number indicating the right edge of the object, in pixels, relative to its parent's right edge. To set this property, call UIObject.move().

Example

The following example moves the check box so it aligns under the right edge of the list box:

```
myCheckbox.move(form.width - listbox.right, myCheckbox.y);
```

UIObject.scaleX

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.scaleX

Description

Property; a number indicating the scaling factor in the *x* direction of the object, relative to its parent.

Example

The following example makes the check box twice as wide and sets the `tmp` variable to the horizontal scale factor:

```
checkbox.scaleX = 200;
var tmp = checkbox.scaleX;
```

UIObject.scaleY

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.scaleY

Description

Property; a number indicating the scaling factor in the *y* direction of the object, relative to its parent.

Example

The following example makes the check box twice as high and sets the `tmp` variable to the vertical scale factor:

```
checkbox.scaleY = 200;
var tmp = checkbox.scaleY;
```

UIObject.setSize()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.setSize(*width*, *height*)

Parameters

width A number that indicates the width of the object, in pixels.

height A number that indicates the height of the object, in pixels.

Returns

Nothing.

Description

Method; resizes the object to the requested size. You should pass only integral values to `UIObject.setSize()`, or the component may appear fuzzy. This method (like all methods and properties of UIObject) is available from any component instance.

When you call this method on a ComboBox instance, the combo box is resized and the `rowHeight` property of the contained list is also changed.

Example

This example resizes the `pBar` component instance to 100 pixels wide and 100 pixels high:

```
pBar.setSize(100, 100);
```

UIObject.setSkin()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.setSkin(id, linkageName)
```

Parameters

id A number indicating the depth of the skin within the component.

linkageName A string indicating an asset in the library.

Returns

A reference to the movie clip (skin) that was attached.

Description

Method; sets a skin in the component instance. Use this method in a component's class file when you are creating a component. For more information, see "About assigning skins" on page 950.

You cannot use this method to set a component's skins at runtime (for example, the way you set a component's styles at runtime).

Example

This example is a code snippet from the class file of a new component, called Shape. It creates a variable, `themeShape` and sets it to the Linkage identifier of the skin. In the `createChildren()` method, the `setSkin()` method is called and passed the id `1` and the variable that holds the linkage identifier of the skin:

```
class Shape extends UIComponent{

  static var symbolName:String = "Shape";
  static var symbolOwner:Object = Shape;
  var className:String = "Shape";
```

```
  var themeShape:String = "circle_skin"

  function Shape(){
  }

  function init(Void):Void{
    super.init();
  }

  function createChildren():Void{
    setSkin(1, themeShape);
    super.createChildren();
  }
}
```

UIObject.setStyle()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.setStyle(*propertyName*, *value*)

Parameters

propertyName A string indicating the name of the style property (for example, "fontWeight", "borderStyle", and so on).

value The value of the property. If the value is a string, it must be enclosed in quotation marks.

Returns

A UIObject object that is an instance of the specified class.

Description

Method; sets the style property on the style declaration or object. If the style property is an inheriting style, the children of the object are notified of the new value.

For a list of the styles supported by each component, see individual component entries. For example, Button component styles are listed in "Using styles with the Button component" under "Button component."

Example

The following code sets the fontWeight style property of the check box instance cb to bold:

```
cb.setStyle("fontWeight", "bold");
```

UIObject.top

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
componentInstance.top
```

Description

Property (read-only); a number indicating the top edge of the object, in pixels, relative to its parent. To set this property, call `UIObject.move()`.

UIObject.unload

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(unload){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.unload = function(eventObject){
   ...
}
componentInstance.addEventListener("unload", listenerObject)
```

Description

Event; notifies listeners that the subobjects of this object are being unloaded.

The first usage example uses an `on()` handler and must be attached directly to a component instance.

The second usage example uses a dispatcher/listener event model. A component instance (*componentInstance*) dispatches an event (in this case, `unload`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. Each event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example deletes `sym1` when the `unload` event is triggered:

```
function unload(){
  form.destroyObject(sym1);
}
form.addEventListener("unload", this);
```

UIObject.visible

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.visible

Description

Property; a Boolean value indicating whether the object is visible (`true`) or not (`false`).

Example

The following example makes the `myLoader` loader instance visible:

```
myLoader.visible = true;
```

UIObject.width

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.width

Description

Property (read-only); a number indicating the width of the object, in pixels. To change the width, call `UIObject.setSize()`.

Example

The following example makes the check box wider:

```
myCheckbox.setSize(myCheckbox.width + 10, myCheckbox.height);
```

UIObject.x

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.x

Description

Property (read-only); a number indicating the left edge of the object, in pixels. To set this property, call `UIObject.move()`.

Example

The following example moves the check box 10 pixels to the right:

```
myCheckbox.move(myCheckbox.x + 10, myCheckbox.y);
```

UIObject.y

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

componentInstance.y

Description

Property (read-only); a number indicating the top edge of the object, in pixels. To set this property, call `UIObject.move()`.

Example

The following example moves the check box down 10 pixels:

```
myCheckbox.move(myCheckbox.x, myCheckbox.y + 10);
```

UIScrollBar component

The UIScrollBar component allows you to add a scroll bar to a text field. You can add a scroll bar to a text field while authoring, or at runtime with ActionScript.

The UIScrollBar component functions like any other scroll bar. It contains arrow buttons at either end and a scroll track and scroll box (thumb) in between. It can be attached to any edge of a text field and used both vertically and horizontally.

Using the UIScrollBar component

To use the UIScrollBar component, verify that object snapping is turned on (View > Snapping > Snap to Objects). Then create a text input field on the Stage and drag the UIScrollBar component from the Components panel to any quadrant of the text field's bounding box.

If the length of the scroll bar is smaller than the combined size of its scroll arrows, it will not be displayed correctly. One of the arrow buttons will become hidden behind the other. Flash does not provide error checking for this. In this case it is a good idea to hide the scroll bar with ActionScript. If the scroll bar is sized so that there is not enough room for the scroll box (thumb), Flash makes the scroll box invisible.

Unlike many other components, the UIScrollBar component can receive continuous mouse input, such as when the user holds the mouse button down, rather than requiring repeated clicks.

There is no keyboard interaction with the UIScrollBar component.

UIScrollBar parameters

You can set the following authoring parameters for each UIScrollBar instance in the Property inspector or in the Component inspector:

_targetInstanceName indicates the name of the text field instance that the UIScrollBar component is attached to.

horizontal indicates whether the scroll bar is oriented horizontally (`true`) or vertically (`false`). The default value is `false`.

You can write ActionScript to control these and additional options for a UIScrollBar component using its properties, methods, and events. For more information, see "UIScrollBar class" on page 833.

Creating an application with the UIScrollBar component

The following procedure explains how to add a UIScrollBar component to an application while authoring.

To create an application with the UIScrollBar component:

1. Create a text input field and give it an instance name in the Property inspector. Add enough text to the field so that users will have to scroll to view it all.
2. In the Property inspector, set the Line Type of the text input field to Multiline, or Multiline No Wrap if you plan to use the scroll bar horizontally.

3. Verify that object snapping is turned on (View > Snapping > Snap to Objects).
4. Drag a UIScrollBar instance from the Components panel onto the text input field near the side you want to attach it to. The component must overlap with the text field when you release the mouse in order for it to be properly bound to the field.

   The `_targetInstanceName` property of the component is automatically populated with the text field instance name in the Property and Component inspectors.
5. Select Control > Test Movie.

   The application runs, and the scroll bar scrolls the contents of the text field.

You can also create a UIScrollBar component instance and associate it with a text field at runtime with ActionScript.

The following code creates a vertically oriented UIScrollBar instance and attaches it to the right side of a text field instance named MyTextField:

```
// createClassObject like any other component. Name it vSB.
createClassObject(mx.controls.UIScrollBar, "vSB", 10);

// set the target text field
vSB.setScrollTarget(MyTextField);

// size it to match the text field
vSB.setSize(16, MyTextField._height);

// move it next to the text field
vSB.move(MyTextField._x + MyTextField._width, MyTextField._y);
```

The following code creates a horizontally oriented UIScrollBar instance and attaches it to the bottom of a text field instance named MyTextField:

```
// createClassObject like any other component. Name it hSB.
_root.createClassObject(mx.controls.UIScrollBar, "hSB", 20);
hSB.horizontal = true

// set the target text field
hSB.setScrollTarget(MyTextField);

// size it to match the text field
hSB.setSize(MyTextField._width, 16);

// move it to the bottom of the text field
hSB.move(MyTextField._x, MyTextField._y + MyTextField._height);
```

Customizing the UIScrollBar component

You can transform a UIScrollBar component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use the `setSize()` method (see `UIObject.setSize()`) or any applicable properties and methods of the UIScrollBar class.

Note, however, that with the Halo theme, the width of a vertically oriented scroll bar must be 16 pixels, and the height of a horizontally oriented scroll bar must also be 16 pixels. These dimensions are determined strictly by the current theme used with the scroll bar. Only the dimension of a scroll bar that corresponds to its length can be changed.

You can customize the appearance of a UIScrollBar instance by using styles and skins.

Using styles with the UIScrollBar component

The UIScrollBar component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`scrollTrackColor`	Sample	The background color for the scroll track.The default value is 0xCCCCCC (light gray).
`symbolColor`	Sample	The color of the up and down scroll arrows. The default value is 0x000000 (black).
`symbolDisabledColor`	Sample	The color of the up and down scroll arrows in a disabled scroll bar. The default value is 0x848384 (dark gray).

Using skins with the UIScrollBar component

The UIScrollBar component uses 13 skins for the track, scroll box (thumb), and buttons. To customize these skin elements, edit the symbols in the Flash UI Components 2/Themes/MMDefault/ScrollBar Assets/States folder. For more information, see "About skinning components" on page 80.

Both horizontal and vertical scroll bars use the same vertical skins, and when displaying a horizontal scroll bar the UIScrollBar component rotates the skins as appropriate.

The UIScrollBar component supports the following skin properties.

Property	Description
`upArrowUpName`	The up (normal) state of the up and left buttons. The default value is `ScrollUpArrowUp`.
`upArrowOverName`	The rollover state of the up and left buttons. The default value is `ScrollUpArrowOver`.
`upArrowDownName`	The pressed state of the up and left buttons. The default value is `ScrollUpArrowDown`.
`downArrowUpName`	The up (normal) state of the down and right buttons. The default value is `ScrollDownArrowUp`.
`downArrowOverName`	The rollover state of the down and right buttons. The default value is `ScrollDownArrowOver`.
`downArrowDownName`	The pressed state of the down and right buttons. The default value is `ScrollDownArrowDown`.

Property	Description
scrollTrackName	The symbol used for the scroll bar's track (background). The default value is ScrollTrack.
scrollTrackOverName	The symbol used for the scroll track (background) when rolled over. The default value is undefined.
scrollTrackDownName	The symbol used for the scroll track (background) when pressed. The default value is undefined.
thumbTopName	The top and left caps of the scroll box (thumb). The default value is ScrollThumbTopUp.
thumbMiddleName	The middle (expandable) part of the thumb. The default value is ScrollThumbMiddleUp.
thumbBottomName	The bottom and right caps of the thumb. The default value is ScrollThumbBottomUp.
thumbGripName	The grip displayed in front of the thumb. The default value is ScrollThumbGripUp.

The following example demonstrates how to put a thin blank line in the middle of the scroll track.

To create movie clip symbols for UIScrollBar skins:

1. Create a new FLA file.
2. Select File > Import > Open External Library, and select the HaloTheme.fla file.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 77.
3. In the theme's Library panel, expand the Flash UI Components 2/Themes/MMDefault folder and drag the ScrollBar Assets folder to the library for your document.
4. Expand the ScrollBar Assets/States folder in the library of your document.
5. Open the symbols you want to customize for editing.

   For example, open the ScrollTrack symbol.
6. Customize the symbol as desired.

   For example, draw a black rectangle in the middle of the track using a 1 x 4 rectangle at (8,0).
7. Repeat steps 5-6 for all symbols you want to customize.

   For example, draw the same line on the ScrollTrackDisabled symbol.
8. Click the Back button to return to the main Timeline.
9. Create an input type TextField instance on the Stage.
10. Drag a UIScrollBar component to the TextField instance.
11. Select Control > Test Movie.

UIScrollBar class

Inheritance MovieClip > UIObject class > UIComponent class > ScrollBar > UIScrollBar

ActionScript Class Name mx.controls.UIScrollBar

The properties of the UIScrollBar class let you adjust the scroll position and the amount of scrolling that occurs when the user clicks the scroll arrows or the scroll track.

Unlike most other components, events are broadcast when the mouse button is pressed and continue broadcasting until the button is released.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.controls.UIScrollBar.version);
```

Note: The code `trace(myUIScrollBarInstance.version);` returns `undefined`.

Method summary for the UIScrollBar class

The following table lists the method of the UIScrollBar class.

Method	Description
`UIScrollBar.setScrollProperties()`	Sets the scroll range of the scroll bar and the size of the text field that the scroll bar is attached to.
`UIScrollBar.setScrollTarget()`	Assigns the scroll bar to a text field.

Methods inherited from the UIObject class

The following table lists the methods the UIScrollBar class inherits from the UIObject class. When calling these methods from the UIScrollBar object, use the form *`UIScrollBarInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.destroyObject()`	Destroys a component instance.
`UIObject.doLater()`	Calls a function when parameters have been set in the Property and Component inspectors.
`UIObject.getStyle()`	Gets the style property from the style declaration or object.
`UIObject.invalidate()`	Marks the object so it will be redrawn on the next frame interval.
`UIObject.move()`	Moves the object to the requested position.
`UIObject.redraw()`	Forces validation of the object so it is drawn in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.

Method	Description
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the UIScrollBar class inherits from the UIComponent class. When calling these methods from the UIScrollBar object, use the form *UIScrollBarInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the UIScrollBar class

The following table lists properties of the UIScrollBar class.

Property	Description
UIScrollBar.lineScrollSize	The number of lines or pixels scrolled when the user clicks the arrow buttons of the scroll bar.
UIScrollBar.pageScrollSize	The number of lines or pixels scrolled when the user clicks the track of the scroll bar.
UIScrollBar.scrollPosition	The current scroll position of the scroll bar.
UIScrollBar._targetInstanceName	The instance name of the text field associated with the UIScrollBar instance.
UIScrollBar.horizontal	A Boolean value indicating whether the scroll bar is oriented vertically (`false`), the default, or horizontally (`true`).

Properties inherited from the UIObject class

The following table lists the properties the UIScrollBar class inherits from the UIObject class. When accessing these properties from the UIScrollBar object, use the form *UIScrollBarInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.

Property	Description
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the UIScrollBar class inherits from the UIComponent class. When accessing these properties from the UIScrollBar object, use the form *`UIScrollBarInstance.propertyName`*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the UIScrollBar class

The following table lists the event of the UIScrollBar class.

Event	Description
UIScrollBar.scroll	Broadcast when any part of the scroll bar is clicked.

Events inherited from the UIObject class

The following table lists the events the UIScrollBar class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.

Event	Description
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the UIScrollBar class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

UIScrollBar.horizontal

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollBarInstance.horizontal

Description

Property; indicates whether the scroll bar is oriented vertically (false) or horizontally (true).

This property can be tested and set. The default value is false.

Example

The following example sets the scroll bar named MyScrollBar to a horizontal orientation:

```
myScrollBar.horizontal = true;
```

UIScrollBar.lineScrollSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollBarInstance.lineScrollSize

Description

Property; gets or sets the number of lines or pixels scrolled when the user clicks the arrow buttons of the UIScrollBar component. If the scroll bar is oriented vertically, the value is a number of lines. If the scroll bar is oriented horizontally, the value is a number of pixels.

The default value is 1.

Example

The following example sets the scroll bar to scroll two lines of text each time the user clicks one of the scroll arrows:

```
myScrollBar.lineScrollSize = 2;
```

UIScrollBar.pageScrollSize

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollBarInstance.pageScrollSize

Description

Property; gets or sets the number of lines or pixels scrolled when the user clicks the track of the UIScrollBar component. If the scroll bar is oriented vertically, the value is a number of lines. If the scroll bar is oriented horizontally, the value is a number of pixels.

You can also set this value by passing a *pageSize* parameter with the `UIScrollBar.setScrollTarget()` method.

Example

The following example sets the scroll bar to scroll 10 lines of text each time the user clicks the scroll track:

```
myScrollBar.pageScrollSize = 10;
```

UIScrollBar.scroll

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(scroll){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.scroll = function(eventObject){
  ...
}
UIScrollBarInstance.addEventListener("scroll", listenerObject)
```

Description

Event; broadcast to all registered listeners when the mouse is clicked (released) over the scroll bar. The `UIScrollBar.scrollPosition` property and the scroll bar's onscreen image are updated before this event is broadcast.

The first usage example uses an `on()` handler and must be attached directly to a UIScrollBar component instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the UIScrollBar component instance `myUIScrollBarComponent`, sends "_level0.myUIScrollBarComponent" to the Output panel:

```
on(scroll){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model, in which the script is placed on a frame in the Timeline that contains the component instance. A component instance (*`UIScrollBarInstance`*) dispatches an event (in this case, `scroll`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event occurs. When the event occurs, it automatically passes an event object (*`eventObject`*) to the listener object method. The event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call `addEventListener()` (see `EventDispatcher.addEventListener()`) on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

In addition to the normal properties of the event object (`type` and `target`), the event object for the `scroll` event includes a third property named `direction`. The `direction` property contains a string describing which way the scroll bar is oriented. The possible values for the `direction` property are `vertical` (the default) and `horizontal`.

For more information about the `type` and `target` event object properties, see "Event objects" on page 415.

Example

The following code implements Usage 1. The code is attached to the UIScrollBar component instance and sends a message to the Output panel when the user clicks the scroll bar. The `on()` handler must be attached directly to the UIScrollBar instance.

```
on(scroll){
  trace("UIScrollBar component was clicked");
}
```

The following example implements Usage 2 and creates a listener object called `myListener` with a `scroll` event handler for the `verticalScroll` instance of the UIScrollBar component:

```
myListener = new Object();
myListener.scroll = function(eventObj){
  // insert code to handle the "scroll" event
}
verticalScroll.addEventListener("scroll", myListener);
```

UIScrollBar.scrollPosition

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

`scrollBarInstance``.scrollPosition`

Description

Property; gets or sets the current scroll position of the scrollable text field. The position of the scroll box (thumb) also updates when a new `scrollPosition` value is set. The value of `scrollPosition` depends on whether the UIScrollBar instance is being used for vertical or horizontal scrolling.

If the UIScrollBar instance is being used for vertical scrolling (the most common use), the value of `scrollPosition` is an integer with a range that begins with 0 and ends with a number that is equal to the total number of lines in the text field divided by the number of lines that can be displayed in the text field simultaneously. If `scrollPosition` is set to a number greater than this range, the text field simply scrolls to the end of the text.

To scroll the text to the first line, set `scrollPosition` to 0.

To scroll the text to the end, set `scrollPosition` to the number of lines of text in the text field minus 1. You can determine the number of lines by retrieving the value of the `maxscroll` property of the text field.

If the UIScrollBar instance is being used for horizontal scrolling, the value of `scrollPosition` is an integer value ranging from 0 to the width of the text field, in pixels. You can determine the width of the text field in pixels by getting the value of the `maxhscroll` property of the text field.

The default value of `scrollPosition` is 0.

Example

The following example scrolls the text field to the beginning of the text it contains:

```
myScrollBar.scrollPosition = 0;
```

The following example scrolls the text field to the end of the text it contains:

```
myScrollBar.scrollPosition = myTextField.maxscroll - 1;
```

UIScrollBar.setScrollProperties()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollBarInstance.setScrollProperties(*pageSize*, *minPos*, *maxPos*)

Parameters

pageSize The number of items that can be viewed in the display area. This parameter sets the size of the text field's bounding box. If the scroll bar is vertical, this value is a number of lines of text; if the scroll bar is horizontal, this value is a number of pixels.

minPos This parameter refers to the lowest numbered line of text when the scroll bar is used vertically, or the lowest numbered pixel in the text field's bounding box when the scroll bar is used horizontally. The value is usually 0.

maxPos This value refers to the highest numbered line of text when the scroll bar is used vertically, or the highest numbered pixel in the text field's bounding box when the scroll bar is used horizontally.

Description

Method; sets the scroll range of the scroll bar and the size of the text field that the scroll bar is attached to. This method is primarily useful when you attach a UIScrollBar component to a text field at runtime (using `UIScrollBar.setScrollTarget()`) rather than while authoring.

The `minPos` and `maxPos` values are used together by the UIScrollBar component to determine the scroll range for the scroll bar and the associated text field.

If you use the `replaceText` method to set the text of the text field, you must use `setScrollProperties()` to cause an update of the scroll bars.

Example

The following example sets up a UIScrollBar component to display 10 lines of text at a time in the text field out of a range of 0 to 99 lines:

```
myScrollBar.setScrollProperties(10, 0, 99);
```

UIScrollBar.setScrollTarget()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollBarInstance.setScrollTarget(*textInstance*)

Parameters

textInstance The text field to assign to the scroll bar.

Description

Method; assigns a UIScrollBar component to a text field instance. If you need to associate a text field and a UIScrollBar component at runtime, use this method.

Example

The following example assigns the UIScrollBar instance named `myScrollBar` to the text field instance named `txt`. The scroll bar is oriented vertically.

```
myScrollBar.setScrollTarget(txt);
```

The following example assigns the UIScrollBar instance named `myScrollBar` to the text field instance named `task_list`. The scroll bar is oriented vertically.

```
myScrollBar.setScrollTarget(task_list, true);
```

UIScrollBar._targetInstanceName

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

scrollBarInstance._targetInstanceName

Description

Property; indicates the instance name of the text field associated with a UIScrollBar component. This property can be tested and set. However, it should not be used to create an association between a text field and a scroll bar. Use `UIScrollBar.setScrollTarget()` instead.

Example

The following example gets the name of the text field instance attached to the scroll bar `MyScrollBar` and sets its border to `true`:

```
var theName = myScrollBar._targetInstanceName;
theName.border = true;
```

Web service classes (Flash Professional only)

The web service classes, which are found in the mx.services package, let you access web services that use Simple Object Access Protocol (SOAP). This API is not the same as the WebServiceConnector component API. The web service API is a set of classes that can you use only in ActionScript code, and is common to various Macromedia products. In contrast, the WebServiceConnector component is an API unique to Flash MX Professional 2004 and provides an ActionScript interface to the visual WebServiceConnector component.

The following table lists the classes in the mx.services package. These classes are closely integrated, so when first learning about this package, you may want to read the information in the order in which it is presented in the table.

Class	Description
WebService class (Flash Professional only)	Using a Web Service Definition Language (WSDL) file that defines the web service, constructs a new WebService object for calling web service methods and handling callbacks from the web service.
PendingCall class (Flash Professional only)	Object returned from a web service method call that you implement to handle the call's results and faults.
Log class (Flash Professional only)	Optional object used to record activity related to a WebService object.
SOAPCall class (Flash Professional only)	Advanced class that contains information about the web service operation, and provides control over certain behaviors.

Making web service classes available at runtime (Flash Professional only)

In order to make the web service classes available at runtime, the WebServiceConnector component must be in your FLA file's library. This component contains the runtime classes that let you work with web services. For details on adding these classes to your FLA file, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Note: These classes are automatically made available to your Flash document when you add a WebServiceConnector component to your FLA.

Log class (Flash Professional only)

ActionScript Class Name mx.services.Log

The Log class is part of the mx.services package and is intended to be used with the WebService class. For an overview of the classes in the mx.services package, see "Web service classes (Flash Professional only)" on page 842.

You can create a new Log object to record activity related to a WebService object. To execute code when messages are sent to a Log object, use the `Log.onLog()` callback function. There is no log file; the logging mechanism is whatever you have used in the `onLog()` callback function, such as sending the log messages to a `trace()` statement.

The constructor for this class creates a Log object that can be passed as an optional parameter to the WebService constructor (see "WebService class (Flash Professional only)" on page 856).

Callback summary for the Log object

The following table lists the callback of the Log object.

Callback	Description
`Log.onLog()`	Called by Flash Player when a log message is sent to a log file.

Constructor for the Log class

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myWebSrvcLog = new Log([logLevel] [, logName]);
```

Parameters

`logLevel` A level to indicate the category of information you want to record in the log. Three log levels are available:

- `Log.BRIEF` The log records primary life-cycle event and error notifications. This is the default value.
- `Log.VERBOSE` The log records all life-cycle event and error notifications.
- `Log.DEBUG` The log records metrics and fine-grained events and errors.

`logName` Optional name that is included with each log message. If you are using multiple Log objects, you can use the log name to determine which log recorded a given message.

Returns

Nothing.

Description

Constructor; creates a Log object. After you create the Log object, you can pass it to a web service to get messages.

Example

You can call the `new Log` constructor to return a Log object to pass to your web service:

```
// creates a new log object
myWebSrvcLog = new Log();
myWebSrvcLog.onLog = function(txt)
{
  myTrace(txt)
}
```

You then pass this Log object as a parameter to the WebService constructor:

```
myWebSrvc = new WebService("http://www.myco.com/info.wsdl", myWebSrvcLog);
```

As the web services code executes and messages are sent to the Log object, the `onLog()` function of your Log object is called. This is the only place to put code that displays the log messages if you want to see them in real time.

The following are examples of log messages:

```
7/30 15:22:43 [INFO] SOAP: Decoding PendingCall response
7/30 15:22:43 [DEBUG] SOAP: Decoding SOAP response envelope
7/30 15:22:43 [DEBUG] SOAP: Decoding SOAP response body
7/30 15:22:44 [INFO] SOAP: Decoded SOAP response into result [16 millis]
7/30 15:22:46 [INFO] SOAP: Received SOAP response from network [6469 millis]
7/30 15:22:46 [INFO] SOAP: Parsed SOAP response XML [15 millis]
7/30 15:22:46 [INFO] SOAP: Decoding PendingCall response
7/30 15:22:46 [DEBUG] SOAP: Decoding SOAP response envelope
7/30 15:22:46 [DEBUG] SOAP: Decoding SOAP response body
7/30 15:22:46 [INFO] SOAP: Decoded SOAP response into result [16 millis]
```

Log.onLog()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myWebSrvcLog.onLog = function(message)
```

Parameters

message The log message passed to the handler.

Returns

Nothing.

Description

Callback function; called by Flash Player when a log message is sent to a log file. This function is a good place to put code that records or displays the log messages, such as a `trace` command. (For information about the structure of the log, see "Log class (Flash Professional only)" on page 842.)

Example

The following example creates a new Log object, passes it to a new WebService object, and handles the log messages.

```
// creates a new Log object
myWebSrvcLog = new Log();
// passes the Log object to the web service
myWebService = new WebService(wsdlURI, myWebSrvcLog);
// handles incoming log messages
myWebSrvcLog.onLog = function(message)
{
```

```
  mytrace("Log Event:\r myWebSrvcLog.message="+message+);
}
```

PendingCall class (Flash Professional only)

ActionScript Class Name mx.services.PendingCall

The PendingCall class is part of the mx.services package and is used with the WebService class. For an overview of the classes in the mx.services package, see "Web service classes (Flash Professional only)" on page 842.

You don't create a PendingCall object or use a constructor function; instead, when you call a method on a WebService object, the WebService method returns a PendingCall object. You use the `PendingCall.onResult` and `PendingCall.onFault` callback functions to handle the asynchronous response from the web service method. If the web service method returns a fault, Flash Player calls `PendingCall.onFault` and passes a SOAPFault object that represents the XML SOAP fault returned by the server or web service. A SOAPFault object is not constructed directly by you, but is returned as the result of a failure. This object is an ActionScript mapping of the SOAPFault XML type.

If the web service invocation is successful, Flash Player calls `PendingCall.onResult` and passes a result object. The result object is the XML response from the web service, decoded or deserialized into ActionScript. For more information about the WebService object, see "WebService class (Flash Professional only)" on page 856.

The PendingCall object also gives you access to multiple output parameters when the web service method returns more than one result. The "return value" referred to in this API is simply the first (or only) result; to gain access to all of the results, you can use the "get output" functions. For example, if the return value delivered to you in the parameter to the `onResult` callback is not the only result you want to access, you can use `getOutputValues()` (which returns an array) or `getOutputValue()` (which returns an individual value) to get the ActionScript decoded values.

You can also access the SOAPParameter object directly. The SOAPParameter object is an ActionScript object with two properties: `value` (the output parameter's ActionScript value) and `element` (the output parameter's XML value). The following functions return a SOAPParameter object, or an array of SOAPParameter objects: `getOutputParameters()`, `getOutputParameterByName()`, and `getOutputParameter()`.

Method summary for the PendingCall class

The following table lists methods of the PendingCall class.

Method	Description
`PendingCall.getOutputParameter()`	Retrieves a SOAPParameter object by index.
`PendingCall.getOutputParameterByName()`	Retrieves a SOAPParameter object by name.
`PendingCall.getOutputParameters()`	Retrieves an array of SOAPParameter objects.

Method	Description
PendingCall.getOutputValue()	Retrieves the output value according to the specified index.
PendingCall.getOutputValues()	Retrieves an array of all the output values.

Property summary for the PendingCall object

The following table lists properties of the PendingCall class.

Property	Description
PendingCall.myCall	The SOAPCall operation descriptor for the PendingCall operation.
PendingCall.request	The SOAP request in raw XML format.
PendingCall.response	The SOAP response in raw XML format.

Callback summary for the PendingCall object

The following table lists the callbacks of the PendingCall class.

Callback	Description
PendingCall.onFault	Called by Flash Player when a web service method has failed and returned an error.
PendingCall.onResult	Called when a method has succeeded and returned a result.

PendingCall.getOutputParameter()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myPendingCall.getOutputParameter(*index*)

Parameters

index The zero-based index of the parameter.

Returns

A SOAPParameter object with two properties: `value` (the output parameter's ActionScript value) and `element` (the output parameter's XML value).

Description

Function; gets an additional output parameter of the SOAPParameter object, which contains the value and the XML element. SOAP RPC calls may return multiple output parameters. The first (or only) return value is always delivered in the *result* parameter of the `onResult` callback function, but to get access to the other return values, you must use functions such as `getOutputParameter()` and `getOutputValue()`. The `getOutputParameter()` function returns the *n*th output parameter as a SOAPParameter object.

Example

Given the SOAP descriptor file below, `getOutputParameter(1)` would return a SOAPParameter object with `value="Hi there!"` and `element=the <outParam2> XMLNode`.

```
...
<SOAP:Body>
  <rpcResponse>
    <outParam1 xsi:type="xsd:int">54</outParam1>
    <outParam2 xsi:type="xsd:string">Hi there!</outParam2>
    <outParam3 xsi:type="xsd:boolean">true</outParam3>
  </rpcResponse>
</SOAP:Body>
...
```

See also

`PendingCall.getOutputParameterByName()`, `PendingCall.getOutputParameters()`, `PendingCall.getOutputValue()`, `PendingCall.getOutputValues()`

PendingCall.getOutputParameterByName()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myPendingCall.getOutputParameterByName(var localName)
```

Parameters

localName The local name of the parameter. In other words, the name of an XML element, stripped of any namespace information. For example, the local name of both of the following elements is `bob`:

```
<bob abc="123">
<xsd:bob def="ghi">
```

Returns

A SOAPParameter object with two properties: `value` (the output parameter's ActionScript value) and `element` (the output parameter's XML value).

Description

Function; gets any output parameter as a SOAPParameter object, which contains the value and the XML element. SOAP RPC calls may return multiple output parameters. The first (or only) return value is always delivered in the *result* parameter of the `onResult` callback function, but to get access to the other return values, you must use functions such as `getOutputParameterByName()`. This function returns the output parameter with the name *localName*.

Example

Given the SOAP descriptor file below, `getOutputParameterByName("outParam2")` would return a SOAPParameter object with `value="Hi there!"` and `element=the <outParam2> XMLNode`.

```
...
<SOAP:Body>
  <rpcResponse>
    <outParam1 xsi:type="xsd:int">54</outParam1>
    <outParam2 xsi:type="xsd:string">Hi there!</outParam2>
    <outParam3 xsi:type="xsd:boolean">true</outParam3>
  </rpcResponse>
</SOAP:Body>
...
```

See also

`PendingCall.getOutputParameter()`, `PendingCall.getOutputParameters()`, `PendingCall.getOutputValue()`, `PendingCall.getOutputValues()`

PendingCall.getOutputParameters()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myPendingCall.getOutputParameters()
```

Parameters

None.

Returns

A SOAPParameter object with two properties: `value` (the output parameter's ActionScript value) and `element` (the output parameter's XML value).

Description

Function; gets additional output parameters of the SOAPParameter object, which contains the values and the XML elements. SOAP RPC calls may return multiple output parameters. The first (or only) return value is always delivered in the *result* parameter of the `onResult` callback function, but to get access to the other return values, you must use functions such as `getOutputParameters()` and `getOutputValues()`.

See also

`PendingCall.getOutputParameterByName()`, `PendingCall.getOutputParameter()`, `PendingCall.getOutputValue()`, `PendingCall.getOutputValues()`

PendingCall.getOutputValue()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myPendingCall`.getOutputValue(var` *index*`)`

Parameters

index The index of an output parameter. The first parameter is index 0.

Returns

The *n*th output parameter.

Description

Function; gets the decoded ActionScript value of an individual output parameter. SOAP RPC calls may return multiple output parameters. The first (or only) return value is always delivered in the *result* parameter of the `onResult` callback function, but to get access to the other return values, you must use functions such as `getOutputValue()` and `getOutputParameter()`. The `getOutputValue()` function returns the *n*th output parameter.

Example

Given the SOAP descriptor file below, `getOutputValue(2)` would return `true`.

```
...
<SOAP:Body>
  <rpcResponse>
    <outParam1 xsi:type="xsd:int">54</outParam1>
    <outParam2 xsi:type="xsd:string">Hi there!</outParam2>
    <outParam3 xsi:type="xsd:boolean">true</outParam3>
  </rpcResponse>
</SOAP:Body>
...
```

See also

PendingCall.getOutputParameterByName(),PendingCall.getOutputParameter(), PendingCall.getOutputParameters(), PendingCall.getOutputValues()

PendingCall.getOutputValues()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

myPendingCall.getOutputValues()

Parameters

None.

Returns

An array of all output parameters' decoded values.

Description

Function; gets the decoded ActionScript value of all output parameters. SOAP RPC calls can return multiple output parameters. The first (or only) return value is always delivered in the *result* parameter of the onResult callback function, but to get access to the other return values, you must use functions such as getOutputValues() and getOutputParameters().

See also

PendingCall.getOutputParameterByName(), PendingCall.getOutputParameter(), PendingCall.getOutputParameters(), PendingCall.getOutputValue()

PendingCall.myCall

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

PendingCall.myCall

Description

Property; the SOAPCall object corresponding to the PendingCall operation. The SOAPCall object contains information about the web service operation, and provides control over certain behaviors. For more information, see "SOAPCall class (Flash Professional only)" on page 854.

Example

The following `onResult` callback traces the name of the SOAPCall operation.

```
callback.onResult = function(result)
{
  // Check my operation name
  trace("My operation name is " + this.myCall.name);
}
```

PendingCall.onFault

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myPendingCallObj.onFault = function(fault)
{
  // your code here
}
```

Parameters

`fault` Decoded ActionScript object version of the SOAPFault object with properties. If the error information came from a server in the form of XML, the SOAPFault object is the decoded ActionScript version of that XML.

The type of error object returned to `PendingCall.onFault` is a SOAPFault object. It is not constructed directly by you, but is returned as the result of a failure. This object is an ActionScript mapping of the SOAPFault XML type.

SOAPFault property	Description
`faultcode`	String; a short string describing the error.
`faultstring`	String; the human-readable description of the error.
`detail`	String; the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.
`element`	XML; the XML object representing the XML version of the fault.
`faultactor`	String; the source of the fault (optional if an intermediary is not involved).

Returns

Nothing.

Description

Callback function; you provide this function, which Flash Player calls when a web service method has failed and returned an error. The *`fault`* parameter is an ActionScript SOAPFault object.

This is a good place to put code that handles any faults, for example, by telling the user that the server isn't available or to contact technical support, if appropriate.

Example

The following example handles errors returned from the web service method.

```
// handles any error returned from the use of a web service method
myPendingCallObj = myWebService.methodName(params)
myPendingCallObj.onFault = function(fault)
{
  // catches the SOAP fault
  DebugOutputField.text = fault.faultstring;

  // add code to handle any faults, for example, by telling the
  // user that the server isn't available or to contact technical
  // support

}
```

PendingCall.onResult

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myPendingCallObj.onResult = function(result)
{
  // your code here

}
```

Parameters

result Decoded ActionScript object version of the XML result returned by a web service method called with `myPendingCallObj = myWebService.methodName(params)`.

Returns

Nothing.

Description

Callback function; you provide this function, which Flash Player calls when a web service method succeeds and returns a result. The result is a decoded ActionScript object version of the XML returned by the operation. In this function, include code that takes appropriate action based on the result. To return the raw XML instead of the decoded result, access the `PendingCall.response` property.

Example

The following example handles results returned from the web service method.

```
// handles results returned from the use of a web service method
myPendingCallObj = myWebService.methodName(params)
myPendingCallObj.onResult = function(result)
{
  // catch the result and handle it for this application
  ResultOutputField.text = result;
}
```

See also

PendingCall.response

PendingCall.request

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
rawXML = myPendingCallback.request;
```

Description

Property; contains the raw XML form of the current request sent with `myPendingCallback = myWebService.methodName()`. Normally, you would not have any use for `PendingCall.request`, but you can use it if you are interested in the SOAP communications that are sent over the network. To get the ActionScript version of the results of the request, use `myPendingCallback.onResult`.

PendingCall.response

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
rawXML = myPendingCallback.response;
```

Description

Property; contains the raw XML form of the response to the most recent web service method call sent with `myPendingCallback = myWebService.methodName()`. Normally, you would not have any use for `PendingCall.response`, but you can use it if you are interested in the SOAP communications that are sent over the network. To get the corresponding ActionScript version of the results of the request, use `myPendingCallback.onResult`.

SOAPCall class (Flash Professional only)

ActionScript Class Name mx.services.SOAPCall

The SOAPCall class is part of the mx.services package and is an advanced class to be used with the WebService class (see "WebService class (Flash Professional only)" on page 856). For an overview of the classes in the mx.data.services package, see "Web service classes (Flash Professional only)" on page 842.

The SOAPCall object is not constructed by you. Instead, when you call a method on a WebService object, the WebService object returns a PendingCall object. To access the associated SOAPCall object, use `myPendingCall.myCall`.

When you create a new WebService object, it contains the methods that correspond to operations in the WSDL URL you pass in. Behind the scenes, a SOAPCall object is created for each operation in the WSDL as well. The SOAPCall object is the descriptor of the operation, and as such contains all the information about that operation (how the XML should look on the network, the operation style, and so on). It also provides control over certain behaviors. You can get the SOAPCall object for a given operation by using the `WebService.getCall()` function. There is a single SOAPCall for each operation, shared by all active calls to that operation. Once you have the SOAPCall object, you can customize the descriptor by doing the following:

- Turning on/off decoding of the XML response
- Turning on/off the delay of converting SOAP arrays into ActionScript objects
- Changing the concurrency configuration for a given operation

Property summary for the SOAPCall object

The following table lists the properties of the SOAPCall object.

Property	Description
`SOAPCall.concurrency`	The number of concurrent requests.
`SOAPCall.doDecoding`	Turns the decoding of the XML response on or off.
`SOAPCall.doLazyDecoding`	Turns "lazy decoding" (the delay of turning SOAP arrays into ActionScript objects) on or off.

SOAPCall.concurrency

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`SOAPCall``.concurrency`

Description

Property; the number of concurrent requests. Possible values are listed in the table below:

Value	Description
`SOAPCall.MULTIPLE_CONCURRENCY`	Allows multiple active calls.
`SOAPCall.SINGLE_CONCURRENCY`	Allows only one call at a time by causing a fault after one is active.
`SOAPCall.LAST_CONCURRENCY`	Allows only one call by cancelling previous ones.

SOAPCall.doDecoding

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`SOAPCall``.doDecoding`

Description

Property; turns decoding of the XML response on (`true`) or off (`false`). By default, the XML response is converted (decoded) into ActionScript objects. If you want just the XML, set `SOAPCall.doDecoding` to `false`.

SOAPCall.doLazyDecoding

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`SOAPCall``.doLazyDecoding`

Description

Property; turns "lazy decoding" of arrays on (`true`) or off (`false`). By default, a "lazy decoding" algorithm is used to delay turning SOAP arrays into ActionScript objects until the last moment; this makes functions execute much more quickly when returning large data sets. This means any arrays you receive from the remote location are ArrayProxy objects. Then when you access a particular index (`foo[5]`), that element is automatically decoded if necessary. You can turn this behavior off (which causes all arrays to be fully decoded) by setting `SOAPCall.doLazyDecoding` to `false`.

WebService class (Flash Professional only)

ActionScript Class Name mx.services.WebService

The WebService class is part of the mx.services package and is used with the Log, PendingCall, and SOAPCall classes. For an overview of the classes in the mx.services package, see "Web service classes (Flash Professional only)" on page 842.

Note: The WebService class is not the same as the WebServiceConnector class. The WebServiceConnector class provides an ActionScript interface to the visual WebServiceConnector component.

The WebService object acts as a local reference to a remote web service. When you create a new WebService object, the WSDL file that defines the web service gets downloaded, parsed, and placed in the object. You can then call the methods of the web service directly on the WebService object and handle any callbacks from the web service. When the WSDL has been successfully processed and the WebService object is ready, the `WebService.onLoad` callback is invoked. If there is a problem loading the WSDL, the `WebService.onFault` callback is invoked.

When you call a method on a WebService object, the return value is a callback object. The object type of the callback returned from all web service methods is `PendingCall`. These objects are normally not constructed by you, but instead are constructed automatically as a result of the *`webServiceObject.webServiceMethodName()`* method that was called. These objects are not the result of the WebService call, which occurs later. Instead, the PendingCall object represents the call in progress. When the WebService operation finishes executing (usually several seconds after a method is called), the various PendingCall data fields are filled in, and the `PendingCall.onResult` or `PendingCall.onFault` callback you provide is called. For more information about the PendingCall object, see "PendingCall class (Flash Professional only)" on page 845.

Flash Player queues any calls you make before the WSDL is parsed, and attempts to execute them after parsing the WSDL. This is because the WSDL contains information that is necessary for correctly encoding and sending a SOAP request. Function calls that you make after the WSDL has been parsed do not need to be queued; they are executed immediately. If a queued call doesn't match the name of any of the operations defined in the WSDL, Flash Player returns a fault to the callback object you were given when you originally made the call.

The WebServices API, included under the mx.services package, consists of the WebService class, the Log class, the PendingCall class, and the PendingCall class.

To make the web service classes available at runtime, you must have the WebServiceConnector component in your FLA file's library. If you're using ActionScript only to access a web service at runtime, you must add this component manually to your document's library. For information on how to add this component to your document, see Chapter 14, "Data Integration (Flash Professional Only)," in *Using Flash*.

Method summary for the WebService object

The following table lists methods of the WebService object.

Method	Description
`WebService.getCall()`	Gets the SOAPCall object for a given operation.
`WebService.myMethodName()`	Invokes a specific web service operation defined by the WSDL.

Callback summary for the WebService object

The following table lists the callbacks of the WebService object.

Callback	Description
`WebService.onFault`	Called when an error occurs during WSDL parsing.
`WebService.onLoad`	Called when the web service has successfully loaded and parsed its WSDL file.

Supported types (Flash Professional only)

The web services classes support a subset of XML schema types (data types) as defined in the tables below.

Complex data types and the SOAP-encoded array type are also supported, and these may be composed of other complex types, arrays, or built-in XML schema types:

- "Numeric Simple types" on page 857
- "Date and Time Simple types" on page 858
- "Name and String Simple types" on page 858
- "Boolean type" on page 858
- "Object types" on page 859
- "Supported XML schema object elements" on page 859

Numeric Simple types

XML schema type	ActionScript binding
`decimal`	Number
`integer`	Number
`negativeInteger`	Number
`nonNegativeInteger`	Number
`positiveInteger`	Number
`long`	Number
`int`	Number
`short`	Number

XML schema type	ActionScript binding
byte	Number
unsignedLong	Number
unsignedShort	Number
unsignedInt	Number
unsignedByte	Number
float	Number
double	Number

Date and Time Simple types

XML schema type	ActionScript binding
date	Date object
datetime	Date object
duration	Date object
gDay	Date object
gMonth	Date object
gMonthDay	Date object
gYear	Date object
gYearMonth	Date object
time	Date object

Name and String Simple types

XML schema type	ActionScript binding
string	ActionScript string
normalizedString	ActionScript string
QName	mx.services.Qname object

Boolean type

XML schema type	ActionScript binding
Boolean	Boolean

Object types

XML schema type	ActionScript binding
`Any`	XML object
`Complex Type`	ActionScript object composed of properties of any supported type
`Array`	ActionScript array composed of any supported object or type

Supported XML schema object elements

The following schema description illustrates the supported XML schema object elements:

```
schema
     complexType
          complexContent
               restriction
          sequence | simpleContent
               restriction
     element
          complexType | simpleType
```

WebService security (Flash Professional only)

The methods and callbacks of the WebService class conform to the Flash Player security model.

User authentication and authorization The authentication and authorization rules are the same for the WebService API as they are for any XML network operation from Flash. SOAP itself does not specify any means of authentication and authorization. For example, when the underlying HTTP transport returns an HTTP BASIC response in the HTTP headers, the browser responds by presenting a dialog box and subsequently attaching the user's input to the HTTP headers in subsequent messages. This mechanism exists at a level lower than SOAP and is part of the Flash HTTP authentication design.

Message integrity Message-level security involves the encryption of the SOAP messages themselves, at a conceptual layer above the network packets on which the SOAP messages are delivered.

Transport security The underlying network transport for Flash Player SOAP web services is always HTTP `POST`. Therefore, any means of security that can be applied at the Flash HTTP transport layer—such as SSL—is supported through web services invocations from Flash. SSL/HTTPS provides the most common form of transport security for SOAP messaging, and use of HTTP BASIC authentication, coupled with SSL at the transport layer, is the most common form of security for websites today.

Constructor for the WebService class

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myWebServiceObject = new WebService(wsdlURI [, logObject]);
```

Parameters

wsdlURI The URI of the web service WSDL file.

logObject An optional parameter specifying the name of the Log object for this web service. If this parameter is used, the Log object must be created first. For more information, see "Log class (Flash Professional only)" on page 842.

Returns

Nothing.

Description

Constructor function; creates a WebService object. When you call `new WebService()`, you provide a WSDL URL, and Flash Player returns a WebService object. The constructor can optionally accept a proxy URI and a Log object.

If you want, you can use two callbacks for the WebService object. Flash Player calls *webServiceObject*`.onLoad` when it finishes parsing the WSDL file and the object is complete. This is a good place to put code you want to execute only after the WSDL file has been completely parsed. For example, you might choose to put your first web service method call in this function.

Flash Player calls *webServiceObject*`.onFault` when an error occurs in finding or parsing the WSDL file. This is a good place to put debugging code and code that tells users that the server is unavailable, that they should try again later, or similar information. For more information, see the individual entries for these functions.

Invoking a web service operation You invoke a web service operation as a method directly available on the web service. For example, if your web service has the method `getCompanyInfo(tickerSymbol)`, you would invoke the method in the following manner:

```
myPendingCallObject = myWebServiceObject.getCompanyInfo(tickerSymbol);
```

In this example, the callback object is named `myPendingCallObject`. All method invocations are asynchronous, and return a callback object of type `PendingCall`. (*Asynchronous* means that the results of the web service call are not available immediately.)

Consider the following call:

```
x = stockService.getQuote("macr");
```

When you make this call, the object `x` is not the result of `getQuote`; it's a PendingCall object. The actual results are only available later, when the web service operation completes. Your ActionScript code is notified by a call to the `onResult` callback function.

Handling the PendingCall object This callback object is a PendingCall object that you use for handling the results and errors from the web service method that was called (see "PendingCall class (Flash Professional only)" on page 845). Here is an example:

```
MyPendingCallObject = myWebServiceObject.myMethodName(param1, ..., paramN);
MyPendingCallObject.onResult = function(result)
```

```
{
  OutputField.text = result
}
MyPendingCallObject.onFault = function(fault)
{
  DebugField.text = fault.faultCode + "," + fault.faultstring;

  // add code to handle any faults, for example, by telling the
  // user that the server isn't available or to contact technical
  // support

}
```

WebService.getCall()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
getCall(var operationName)
```

Parameters

operationName The web service operation of the corresponding SOAPCall object that you want to retrieve.

Returns

A SOAPCall object.

Description

Function; when you create a new WebService object, it contains the methods corresponding to operations in the WSDL URL you pass in. Behind the scenes, a SOAPCall object is created for each operation in the WSDL as well. The SOAPCall object is the descriptor of the operation, and as such contains all the information about that operation (how the XML should look on the network, the operation style, and so on). It also provides control over certain behaviors. You can get the SOAPCall object for a given operation by using the `getCall()` method. There is a single SOAPCall object for each operation, shared by all active calls to that operation. Once you have the SOAPCall object, you can change the operator descriptor by using the SOAPCall class; see "SOAPCall class (Flash Professional only)" on page 854.

WebService.*myMethodName*()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
callbackObj = myWebServiceObject.myMethodName(param1, ... paramN);
```

Parameters

param1, ... paramN Various parameters, depending on the web service method that is called.

Returns

A PendingCall object to which you can attach a function for handling results and errors on the invocation. For more information, see "PendingCall class (Flash Professional only)" on page 845.

The callback invoked when the response comes back from the WebService method is `PendingCall.onResult` or `PendingCall.onFault`. By uniquely identifying your callback objects, you can manage multiple `onResult` callbacks, as in the following example:

```
myWebService = new WebService("http://www.myCompany.com/myService.wsdl");
callback1 = myWebService.getWeather("02451");
callback1.onResult = function(result)
{
  // do something
}
callback2 = myWebService.getDetailedWeather("02451");
callback2.onResult = function(result)
{
  // do something else
}
```

Description

Method; invokes a web service operation. You invoke the method directly on the web service. For example, if your web service has the method `getCompanyInfo(tickerSymbol)`, you would make the following call:

```
myCallbackObject.myservice.getCompanyInfo(tickerSymbol);
```

All invocations are asynchronous, and return a callback object of type `PendingCall`.

WebService.onFault

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

```
Usage
MyWebServiceObject.onFault = function(fault)
{
  // your code here
}
```

Parameters

fault Decoded ActionScript object version of the error with properties. If the error information came from a server in the form of XML, then the SOAPFault object will be the decoded ActionScript version of that XML.

The type of error object returned to `WebService.onFault` methods is a `SOAPFault` object. This object is not constructed directly by you, but returned as the result of a failure. This object is an ActionScript mapping of the SOAPFault XML type.

SOAPFault property	Description
`faultcode`	String; the short standard QName describing the error.
`faultstring`	String; the human-readable description of the error.
`detail`	String; the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.
`element`	XML; the XML object representing the XML version of the fault.
`faultactor`	String; the source of the fault. Optional if an intermediary is not involved.

Returns

Nothing.

Description

Callback function; called by Flash Player when the `new WebService()` constructor has failed and returned an error. This can happen when the WSDL file cannot be parsed or the file cannot be found. The *fault* parameter is an ActionScript SOAPFault object.

Example

The following example handles any error returned from the creation of the WebService object.

```
MyWebServiceObject.onFault = function(fault)
{
  // captures the fault
  DebugOutputField.text = fault.faultstring;

  // add code to handle any faults, for example, by telling the
  // user that the server isn't available or to contact technical
  // support

}
```

WebService.onLoad

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
myService.onLoad = function(wsdlDocument)
{
  // your code here
}
```

Parameters

wsdlDocument The WSDL XML document.

Returns

Nothing.

Description

Callback function; called by Flash Player when the WebService object has successfully loaded and parsed its WSDL file. If operations are invoked in an application before this callback function is called, they are queued internally and not actually transmitted until the WSDL has loaded.

Example

The following example specifies the WSDL URL, creates a new web service object, and receives the WSDL document after loading.

```
// specify the WSDL URL
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// creates a new web service object
stockService = new WebService(wsdlURI);

// receives the WSDL document after loading
stockService.onLoad = function(wsdlDocument);
{
  // code to execute when the WSDL loading is complete and the
  // object has been created
}
```

WebServiceConnector component (Flash Professional only)

The WebServiceConnector component lets you access remote methods exposed by a server using the industry-standard Simple Object Access Protocol (SOAP). A web service method may accept parameters and return a result. Using the Flash MX Professional 2004 authoring tool and the WebServiceConnector component you can inspect, access, and bind data between a remote web service and your Flash application.

A single instance of a WebServiceConnector component can be used to make multiple calls to the same operation. You need to use a different instance of WebServiceConnector for each different operation you want to call.

Note: The WebServiceConnector component appears on the Stage during application authoring but is not visible in the runtime application.

For introductory information on working with the results of this component, see "Working with schemas in the Schema tab (Flash Professional only)" in *Using Flash*.

Using the WebServiceConnector component (Flash Professional only)

You can use the WebServiceConnector component to connect to a web service and make the properties of the web service available for binding to properties of UI components in your application. To connect to a web service, you must first enter the URL for the WSDL file that represents the web service. You can enter this URL in the Component inspector or the Web Services panel. See "Connecting to web services with the WebService connector component (Flash Professional only)" in *Using Flash*.

For more information on connecting to web services, see "Data binding (Flash Professional only)"in *Using Flash*.

WebServiceConnector parameters

You can set the following authoring parameters for each WebServiceConnector component instance by using the Parameters tab of the Component inspector:

multipleSimultaneousAllowed is a Boolean value that indicates whether multiple calls can take place at the same time; the default value is false. If this parameter is `false`, the `trigger()` method does not perform a call if a call is already in progress. A status event is emitted, with the code `CallAlreadyInProgress`. If this parameter is `true`, the call takes place.

operation is a string indicating the name of an operation that appears within the SOAP port in a WSDL file.

suppressInvalidCalls is a Boolean value that indicates whether to suppress a call if parameters are invalid; the default value is `false`. If this parameter is `true`, the `trigger()` method does not perform a call if the databound parameters fail the validation. A status event is emitted, with the code `InvalidParams`. If this parameter is `false`, the call takes place, using the invalid data as required.

WSDLURL (String type) is the URL of the WSDL file that defines the web service operation. When you set this URL during authoring, the WSDL file is immediately fetched and parsed. The resulting parameters and results information can be seen in the Schema tab of the Component inspector. The service description is also added to the Web Service panel. For example, see www.flash-mx.com/mm/tips/tips.cfc?wsdl.

Common workflow for the WebServiceConnector component

The following procedure shows the typical workflow for the WebServiceConnector component.

To use a WebServiceConnector component:

1. Use the Web Services panel to enter the URL for a web service WSDL file.
2. Add a call to a method of the web service by selecting the method, right-clicking (Windows) or Control-clicking (Macintosh), and selecting Add Method Call from the context menu.

   This creates a WebServiceConnector component instance in your application. The schema for the component can then be found on the Schema tab of the Component inspector. You are free to edit this schema as needed—for example, to provide additional formatting or validation settings.

   Note: The schema for the `params` and `results` component properties is updated each time you change the `WSDLURL` or `operation` parameter. This overwrites any settings that you have edited.

3. Use the Bindings tab in the Component inspector to bind the web service parameters and results that are now defined in your schema to UI components in your application.
4. Add a trigger to initiate the data binding operation in one of the following ways:
   - Attach the Trigger Data Source behavior to a button.
   - Add your own ActionScript to call the `trigger()` method on the WebServiceConnector component.
   - Create a binding between a web service parameter and a UI control, and set its Kind property to AutoTrigger. For more information, see "Schema kinds" in *Using Flash*.

For a step-by-step example that connects and displays a web service using the WebServiceConnector component, see "Web Service Tutorial: Macromedia Tips".

WebServiceConnector class (Flash Professional only)

Inheritance RPC > WebServiceConnector

ActionScript Class Name mx.data.components.WebServiceConnector

This class allows you to connect to remote web services using ActionScript code instead of component instances on the Stage. To use the WebServiceConnector class, you need to add an instance of the WebServiceConnector component to your library. The component does not need to be placed directly on the Stage. You must import the ActionScript class mx.data.components.WebServiceConnector at the beginning of the script or use the fully qualified class name throughout your code.

Method summary for the WebServiceConnector class

The following table lists the method of the WebServiceConnector class.

Method	Description
`WebServiceConnector.trigger()`	Initiates a call to a web service.

Property summary for the WebServiceConnector class

The following table lists properties of the WebServiceConnector class.

Property	Description
`WebServiceConnector.multipleSimultaneousAllowed`	Indicates whether multiple calls can take place at the same time.
`WebServiceConnector.operation`	Indicates the name of an operation that appears within the SOAP port in a WSDL file.
`WebServiceConnector.params`	Specifies data that will be sent to the server when the next `trigger()` operation is executed.
`WebServiceConnector.results`	Identifies data that was received from the server as a result of the `trigger()` operation.
`WebServiceConnector.suppressInvalidCalls`	Indicates whether to suppress a call if parameters are invalid.
`WebServiceConnector.timeout`	Specifies a time period (in seconds) in which the web service connection will fail if results do not come back.
`WebServiceConnector.WSDLURL`	Specifies the URL of the WSDL file that defines the web service operation.

Event summary for the WebServiceConnector class

The following table lists events of the WebServiceConnector class.

Event	Description
`WebServiceConnector.result`	Broadcast when a call to a web service completes successfully.
`WebServiceConnector.send`	Broadcast when the `trigger()` method is in process, after the parameter data has been gathered but before the data is validated and the call to the web service is initiated.
`WebServiceConnector.status`	Broadcast when a call to a web service is initiated, to inform the user of the status of the operation.

WebServiceConnector.multipleSimultaneousAllowed

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.multipleSimultaneousAllowed
```

Description

Property; indicates whether multiple calls can (`true`) or cannot (`false`) take place at the same time. If this property is `false`, the `WebServiceConnector.trigger()` method performs a call if another call is already in progress. A `status` event is emitted, with the code `CallAlreadyInProgress`. If this property is `true`, the call takes place.

When multiple calls are simultaneously in progress, there is no guarantee that they will complete in the order in which they were triggered. Also, the browser and/or operating system may place limits on the number of simultaneous network operations. The most likely limit you may encounter is the browser enforcing a maximum number of URLs that can be downloaded simultaneously. This is something that is often configurable in a browser. However, even in this case, the browser should queue streams, and this should not interfere with the expected behavior of the Flash application.

Example

The following example enables multiple simultaneous calls to `myXmlUrl` to take place:

```
myXmlUrl.multipleSimultaneousAllowed = true;
```

WebServiceConnector.operation

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.operation;
```

Description

Property; the name of an operation that appears within the SOAP port in a WSDL file.

Example

This example returns data from a remote web service and traces a tip and how long the service takes to return the data to the SWF file. Drag a WebServiceConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.WebServiceConnector;

var startTime:Number;
var wscListener:Object = new Object();
wscListener.result = function(evt:Object) {
var resultTimeMS:Number = getTimer()-startTime;
trace("result loaded in "+resultTimeMS+" ms.");
trace(evt.target.results);
};
wscListener.send = function(evt:Object) {
startTime = getTimer();
};
var wsConn:WebServiceConnector = new WebServiceConnector();
wsConn.addEventListener("result", wscListener);
wsConn.addEventListener("send", wscListener);
wsConn.WSDLURL = "http://www.flash-mx.com/mm/tips/tips.cfc?wsdl";
wsConn.operation = "getTipByProduct";
wsConn.params = ["Flash"];
wsConn.suppressInvalidCalls = true;
wsConn.multipleSimultaneousAllowed = false;
wsConn.trigger();
```

WebServiceConnector.params

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.params

Description

Property; specifies data that will be sent to the server when the next `trigger()` operation is executed. The data type is determined by the WSDL description of the web service.

When you call web service methods, the data type of the `params` property must be an ActionScript object or array as follows:

- If the web service is in document format, the data type of `params` is an XML document.
- If you use the Property inspector or Component inspector to set the WSDL URL and operation while authoring, you can provide `params` as an array of parameters in the same order as required by the web service method, such as `[1, "hello", 2432]`.

Example

The following example sets the `params` property for a web service component named `wsc`:

```
wsc.params = [param_txt.text];
```

WebServiceConnector.result

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("result", myListenerObject)
```

Description

Event; broadcast when a call to a web service completes successfully.

The parameter to the event handler is an object with the following fields:

- `type`: the string `"result"`
- `target`: a reference to the object that emitted the event (for example, a WebServiceConnector component)

You can retrieve the actual result value using the `results` property.

Example

The following example defines a function `res` for the `result` event and assigns the function to the `addEventListener` event handler:

```
var res = function (ev) {
trace(ev.target.results);
};
wsc.addEventListener("result", res);
```

This example returns data from a remote web service and traces a tip. Drag a WebServiceConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.WebServiceConnector;
var wscListener:Object = new Object();
wscListener.result = function(evt:Object) {
  trace(evt.target.results);
};
var wsConn:WebServiceConnector = new WebServiceConnector();
wsConn.addEventListener("result", wscListener);
wsConn.WSDLURL = "http://www.flash-mx.com/mm/tips/tips.cfc?wsdl";
wsConn.operation = "getTipByProduct";
wsConn.params = ["Flash"];
wsConn.trigger();
```

WebServiceConnector.results

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.results
```

Description

Property; identifies data that was received from the server as a result of a `trigger()` operation. Each WebServiceConnector component defines how this data is fetched, and what the valid types are. This data appears when the RPC operation has successfully completed, as signaled by the `result` event. It is available until the component is unloaded, or until the next RPC operation.

It is possible for the returned data to be very large. You can manage this in two ways:

- Select an appropriate movie clip, Timeline, or screen as the parent for the WebServiceConnector component. The component's storage memory will become available for garbage collection when the parent is destroyed.
- In ActionScript, you can assign `null` to this property at any time.

WebServiceConnector.send

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("send", myListenerObject)
```

Description

Event; broadcast during the processing of a `trigger()` operation, after the parameter data has been gathered but before the data is validated and the call to the web service is initiated. This is a good place to put code that will modify the parameter data before the call.

The parameter to the event handler is an object with the following fields:

- `type`: the string `"send"`
- `target`: a reference to the object that emitted the event (for example, a WebServiceConnector component)

You can retrieve or modify the actual parameter values by using the `params` property.

Example

The following example defines a function `sendFunction` for the `send` event and assigns the function to the `addEventListener` event handler:

```
var sendFunction = function (sendEnv) {
sendEnv.target.params = [newParam_txt.text];
};
wsc.addEventListener("send", sendFunction);
```

WebServiceConnector.status

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance`.addEventListener("status", `*myListenerObject*`)`

Description

Event; broadcast when a call to a web service is initiated, to inform the user of the status of the operation.

The parameter to the event handler is an object with the following fields:

- `type`: the string `"status"`
- `target`: a reference to the object that emitted the event (for example, a WebServiceConnector component)
- `code`: a string giving the name of the condition that occurred
- `data`: an object whose contents depend on the code

The following are the codes and associated data available for the `status` event:

Code	Data	Description
StatusChange	{callsInProgress:nnn}	This event is emitted whenever a web service call starts or finishes. The item *nnn* gives the number of calls currently in progress.
CallAlreadyInProgress	No data	This event is emitted if `trigger()` is called, `multipleSimultaneousAllowed` is `false`, and a call is already in progress. After this event occurs, the attempted call is considered complete, and there is no `result` or `send` event.
InvalidParams	No data	This event is emitted if the `trigger()` method found that the `params` property did not contain valid data. If the `suppressInvalidCalls` property is `true`, the attempted call is considered complete, and there is no `result` or `send` event.

Here are the possible web service faults:

faultcode	faultstring	detail
Timeout	Timeout while calling method xxx	
MustUnderstand	No callback for header xxx	
Server.Connection	Unable to connect to endpoint: xxx	
VersionMismatch	Request implements version: xxx Response implements version yyy	
Client.Disconnected	Could not load WSDL	Unable to load WSDL, if currently online, please verify the URI and/or format of the WSDL xxx
Server	Faulty WSDL format	Definitions must be the first element in a WSDL document
Server.NoServicesInWSDL	Could not load WSDL	No elements found in WSDL at xxx
WSDL.UnrecognizedNamespace	The WSDL parser had no registered document for the namespace xxxx	
WSDL.UnrecognizedBindingName	The WSDL parser couldn't find a binding named xxx in namespace yyy	
WSDL.UnrecognizedPortTypeName	The WSDL parser couldn't find a portType named xxx in namespace yyy	
WSDL.UnrecognizedMessageName	The WSDL parser couldn't find a message named xxx in namespace yyy	
WSDL.BadElement	Element xxx not resolvable	
WSDL.BadType	Type xxx not resolvable	
Client.NoSuchMethod	Couldn't find method 'xxx' in service	
yyy	yyy - errors reported from server, this depends on which server you talk to	
No.WSDLURL.Defined	The WebServiceConnector component had no WSDL URL defined	

faultcode	faultstring	detail
Unknown.Call.Failure	WebService invocation failed for unknown reasons	
Client.Disconnected	Could not load imported schema	Unable to load schema; if currently online, please verify the URI and/or format of the schema at (XXXX)

Example

The following example defines a function `statusFunction` for the `status` event and assigns the function to the `addEventListener` event handler:

```
var statusFunction = function (stat) {
trace(stat.code);
trace(stat.data.faultcode);
trace(stat.data.faultstring);
};
wsc.addEventListener("status", statusFunction);
```

WebServiceConnector.suppressInvalidCalls

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.suppressInvalidCalls

Description

Property; indicates whether to suppress a call if parameters are invalid. If this property is `true`, the `trigger()` method does not perform a call if the bound parameters fail the validation. A `status` event is emitted, with the code `InvalidParams`. If this property is `false`, the call takes place, using the invalid data as required.

Example

This example displays an error because the required parameters are not being passed. Drag a WebServiceConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.WebServiceConnector;
var res:Function = function (evt:Object) {
  trace(evt.target.results);
};
var stat:Function = function (error:Object) {
  switch (error.code) {
  case 'InvalidParams' :
    trace("Unable to connect to remote Web Service: "+error.code);
    break;
```

```
    case 'StatusChange' :
      break;
    default :
      trace("Error: "+error.code);
      break;
    }
};
var wsConn:WebServiceConnector = new WebServiceConnector();
wsConn.addEventListener("result", res);
wsConn.addEventListener("status", stat);
wsConn.WSDLURL = "http://www.flash-mx.com/mm/tips/tips.cfc?wsdl";
wsConn.operation = "getTipByProduct";
// wsConn.params = ["Flash"];
wsConn.suppressInvalidCalls = true;
wsConn.trigger();
```

To display a tip instead of an error, uncomment the line `wsConn.params = ["Flash"];`.

WebServiceConnector.timeout

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.timeout

Description

Property; a time period, in seconds, in which the web service connection fails if results do not come back. A `status` event (inherited from the WebServiceConnector component) is emitted, with the code `WebServiceFault`, fault code `Timeout`.

WebServiceConnector.trigger()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.trigger();

Description

Method; initiates a call to a web service. Each web service defines exactly what this involves. If the operation is successful, the results of the operation will appear in the `results` property for the web service.

The `trigger()` method performs the following steps:

1. If any data is bound to the `params` property, the method executes all the bindings to ensure that up-to-date data is available. This also causes data validation to occur.
2. If the data is not valid and `suppressInvalidCalls` is set to `true`, the operation is discontinued.
3. If the operation continues, the `send` event is emitted.
4. The actual remote call is initiated using the connection method indicated (for example, HTTP).

Example

This example returns data from a remote web service and traces a tip. Drag a WebServiceConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.WebServiceConnector;
var res:Function = function (evt:Object) {
  trace(evt.target.results);
};
var wsConn:WebServiceConnector = new WebServiceConnector();
wsConn.addEventListener("result", res);
wsConn.WSDLURL = "http://www.flash-mx.com/mm/tips/tips.cfc?wsdl";
wsConn.operation = "getTipByProduct";
wsConn.params = ["Flash"];
wsConn.suppressInvalidCalls = true;
wsConn.trigger();
```

WebServiceConnector.WSDLURL

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.WSDLURL

Description

Property; the URL of the WSDL file that defines the web service operation. When you set this URL while authoring, the WSDL file is immediately fetched and parsed. The resulting parameters and results appear in the Schema tab of the Component inspector. The service description also appears in the Web Service panel.

Example

This example returns data from a remote web service and traces a tip. Drag a WebServiceConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.WebServiceConnector;
```

```
var res:Function = function (evt:Object) {
  trace(evt.target.results);
};
var wsConn:WebServiceConnector = new WebServiceConnector();
wsConn.addEventListener("result", res);
wsConn.WSDLURL = "http://www.flash-mx.com/mm/tips/tips.cfc?wsdl";
wsConn.operation = "getTipByProduct";
wsConn.params = ["Flash"];
wsConn.suppressInvalidCalls = true;
wsConn.trigger();
```

Window component

A Window component displays the contents of a movie clip inside a window with a title bar, a border, and an optional close button.

A Window component can be modal or nonmodal. A modal window prevents mouse and keyboard input from going to other components outside the window. The Window component also supports dragging; a user can click the title bar and drag the window and its contents to another location. Dragging the borders doesn't resize the window.

A live preview of each Window instance reflects changes made to all parameters except contentPath in the Property inspector or Component inspector during authoring.

When you add the Window component to an application, you can use the Accessibility panel to make it accessible to screen readers. First, you must add the following line of code to enable accessibility:

```
mx.accessibility.WindowAccImpl.enableAccessibility();
```

You enable accessibility for a component only once, regardless of how many instances you have of the component. For more information, see Chapter 17, "Creating Accessible Content," in *Using Flash*.

Using the Window component

You can use a window in an application whenever you need to present a user with information or a choice that takes precedence over anything else in the application. For example, you might need a user to fill out a login window, or a window that changes and confirms a new password.

There are several ways to add a window to an application. You can drag a window from the Components panel to the Stage. You can also call `createClassObject()` (see `UIObject.createClassObject()`) to add a window to an application. The third way of adding a window to an application is to use the PopUpManager class. Use the Popup Manager to create modal windows that overlap other objects on the Stage. For more information, see "Window class" on page 882.

If you use the Popup Manager to add a Window component to a document, the Window instance will have its own Focus Manager, distinct from the rest of the document. If you don't use the Popup Manager, the window's contents participate in focus ordering with the other components in the document. For more information about controlling focus, see "Creating custom focus navigation" on page 50 or "FocusManager class" on page 419.

Window parameters

You can set the following authoring parameters for each Window component instance in the Property inspector or in the Component inspector:

contentPath specifies the contents of the window. This can be the linkage identifier of the movie clip or the symbol name of a screen, form, or slide that contains the contents of the window. This can also be an absolute or relative URL for a SWF or JPEG file to load into the window. The default value is `""`. Loaded content is clipped to fit the window.

title indicates the title of the window.

closeButton indicates whether a close button is displayed (`true`) or not (`false`). Clicking the close button broadcasts a `click` event, but doesn't close the window. You must write a handler that calls `Window.deletePopUp()` to explicitly close the window. For more information about the `click` event, see `Window.click`.

Note: If a window was created by means other than the PopUp Manager, you can't close it.

You can write ActionScript to control these and additional options for the Window component using its properties, methods, and events. For more information, see "Window class" on page 882.

Creating an application with the Window component

The following procedure explains how to add a Window component to an application. In this example, the window asks a user to change her password and confirm the new password.

To create an application with the Window component:

1. Create a new movie clip that contains password and password confirmation fields, and OK and Cancel buttons. Name the movie clip **PasswordForm**.

   This is the content that will fill the window. The content should be aligned at 0,0 because it is positioned in the upper left corner of the window.
2. In the library, select the PasswordForm movie clip and select Linkage from the Library options menu.
3. Select Export for ActionScript.

   The linkage identifier **PasswordForm** is automatically entered in the Identifier box.
4. Enter **mx.core.View** in the class field and click OK.
5. Drag a Window component from the Components panel to the Stage and delete the component from the Stage. This adds the component to the library.
6. In the library, select the Window SWC file and select Linkage from the Library options menu.
7. Select Export for ActionScript.
8. Drag a button component from the Components panel to the Stage; in the Property inspector, give it the instance name **button**.
9. Open the Actions panel, and enter the following click handler on Frame 1:

```
buttonListener = new Object();
buttonListener.click = function(){
  myWindow = mx.managers.PopUpManager.createPopUp(_root,
  mx.containers.Window, true, {title:"Change Password",
  contentPath:"PasswordForm"});
  myWindow.setSize(240,110);
}
button.addEventListener("click", buttonListener);
```

This handler calls `PopUpManager.createPopUp()` to instantiate a Window component with the title bar "Change Password"; the window displays the contents of the PasswordForm movie clip when the button is clicked. To close the window when the OK or Cancel button is clicked, you must write another handler.

Customizing the Window component

You can transform a Window component horizontally and vertically while authoring and at runtime. While authoring, select the component on the Stage and use the Free Transform tool or any of the Modify > Transform commands. At runtime, use `UIObject.setSize()`.

Resizing the window does not change the size of the close button or title caption. The title caption is aligned to the left and the close bar to the right.

Using styles with the Window component

A Window component supports the following styles:

Style	Theme	Description
`themeColor`	Halo	The base color scheme of a component. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`. The default value is `"haloGreen"`.
`backgroundColor`	Both	The background color. The default value is white for the Halo theme and 0xEFEBEF (light gray) for the Sample theme.
`border styles`	Both	The Window component uses a RectBorder instance as its border and responds to the styles defined on that class. See "RectBorder class" on page 647. The Window component has a component-specific border style of "`default`" with the Halo theme and "`outset`" with the Sample theme.
`color`	Both	The text color. The default value is 0x0B333C for the Halo theme and blank for the Sample theme.
`disabledColor`	Both	The color for text when the component is disabled. The default color is 0x848384 (dark gray).
`embedFonts`	Both	A Boolean value that indicates whether the font specified in `fontFamily` is an embedded font. This style must be set to `true` if `fontFamily` refers to an embedded font. Otherwise, the embedded font will not be used. If this style is set to `true` and `fontFamily` does not refer to an embedded font, no text will be displayed. The default value is `false`.
`fontFamily`	Both	The font name for text. The default value is `"_sans"`.
`fontSize`	Both	The point size for the font. The default value is 10.
`fontStyle`	Both	The font style: either `"normal"` or `"italic"`. The default value is `"normal"`.

Style	Theme	Description
fontWeight	Both	The font weight: either "none" or "bold". The default value is "none". All components can also accept the value "normal" in place of "none" during a setStyle() call, but subsequent calls to getStyle() will return "none".
textAlign	Both	The text alignment: either "left", "right", or "center". The default value is "left".
textDecoration	Both	The text decoration: either "none" or "underline". The default value is "none".
textIndent	Both	A number indicating the text indent. The default value is 0.

Text styles can be set on the Window component itself, or they can be set on the _global.styles.windowStyles class style declaration. This has the advantage of not causing style settings to propagate to child components through style inheritance.

The following example demonstrates how to italicize the title of a Window component without having this setting propagate to child components.

```
import mx.containers.Window;
_global.styles.windowStyles.setStyle("fontStyle", "italic");
createClassObject(Window, "window", 1, {title: "A Window"});
```

Notice that this example sets the property before instantiating the window through createClassObject(). For the styles to take effect, they must be set before the window is created.

Using skins with the Window component

The Window component uses skins for its title background and close button, and a RectBorder instance for the border. The Window skins are found in the Flash UI Components 2/Themes/MMDefault/Window Assets folder in each of the theme files. For more information about skinning, see "About skinning components" on page 80. For more information about the RectBorder class and using it to customize borders, see "RectBorder class" on page 647.

The title background skin is always displayed. The height of the background is determined by the skin graphics. The width of the skin is set by the Window component according to the Window instance's size. The close skins are displayed when the closeButton property is set to true and when a change state results from user interaction.

A Window component uses the following skin properties:

Property	Description
skinTitleBackground	The title bar. The default value is TitleBackground.
skinCloseUp	The close button. The default value is CloseButtonUp.
skinCloseDown	The close button it its down state. The default value is CloseButtonDown.

Property	Description
`skinCloseDisabled`	The close button in its disabled state. The default value is `CloseButtonDisabled`.
`skinCloseOver`	The close button in its over state. The default value is `CloseButtonOver`.

The following example demonstrates how to create a new movie clip symbol to use as the title background.

To set the title of an Window component to a custom movie clip symbol:

1. Create a new FLA file.
2. Create a new symbol by selecting Insert > New Symbol.
3. Set the name to `TitleBackground`.
4. If the advanced view is not displayed, click the Advanced button.
5. Select Export for ActionScript.
6. The identifier will be automatically filled out with `TitleBackground`.
7. Set the AS 2.0 class to `mx.skins.SkinElement`.

   SkinElement is a simple class that can be used for all skin elements that don't provide their own ActionScript impelmentation. It provides movement and sizing functionality required by the version 2 component framework.
8. Ensure that Export in First Frame is already selected, and click OK.
9. Open the new symbol for editing.
10. Use the drawing tools to create a box with a red fill and black line.
11. Set the border style to hairline.
12. Set the box, including the border, so that it is positioned at (0,0) and has a width of 100 and height of 22.

    The Window component will set the proper width of the skin as needed but it will use the existing height as the height of the title.
13. Click the Back button to return to the main Timeline.
14. Drag the Window component to the Stage.
15. Select Control > Test Movie.

Window class

Inheritance MovieClip > UIObject class > UIComponent class > View > ScrollView > Window

ActionScript Class Name mx.containers.Window

The properties of the Window class let you do the following at runtime: set the title caption, add a close button, and set the display content. Setting a property of the Window class with ActionScript overrides the parameter of the same name set in the Property inspector or Component Inspector panel.

The best way to instantiate a window is to call `PopUpManager.createPopUp()`. This method creates a window that can be modal (overlapping and disabling existing objects in an application) or nonmodal. For example, the following code creates a modal Window instance (the last parameter indicates modality):

```
var newWindow = PopUpManager.createPopUp(this, Window, true);
```

Flash simulates modality by creating a large transparent window underneath the Window component. Because of the way transparent windows are rendered, you may notice a slight dimming of the objects under the transparent window. You can set the effective transparency by changing the `_global.style.modalTransparency` value from 0 (fully transparent) to 100 (opaque). If you make the window partially transparent, you can also set the color of the window by changing the Modal skin in the default theme.

If you use `PopUpManager.createPopUp()` to create a modal window, you must call `Window.deletePopUp()` to remove it to so that the transparent window is also removed. For example, if you use the close button in the window, you would write the following code:

```
obj.click = function(evt){
   this.deletePopUp();
}
window.addEventListener("click", obj);
```

Note: Code does not stop executing when a modal window is created. In other environments (for example, Microsoft Windows), if you create a modal window, the lines of code that follow the creation of the window do not run until the window is closed. In Flash, the lines of code are run after the window is created and before it is closed.

Each component class has a `version` property, which is a class property. Class properties are available only on the class itself. The `version` property returns a string that indicates the version of the component. To access this property, use the following code:

```
trace(mx.containers.Window.version);
```

Note: The code `trace(myWindowInstance.version);` returns `undefined`.

Method summary for the Window class

The following table lists the method of the Window class.

Method	Description
`Window.deletePopUp()`	Removes a window instance created by `PopUpManager.createPopUp()`.

Methods inherited from the UIObject class

The following table lists the methods the Window class inherits from the UIObject class. When calling these methods from the Window object, use the form *`WindowInstance.methodName`*.

Method	Description
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.createObject()`	Creates a subobject on an object.

Method	Description
UIObject.destroyObject()	Destroys a component instance.
UIObject.doLater()	Calls a function when parameters have been set in the Property and Component inspectors.
UIObject.getStyle()	Gets the style property from the style declaration or object.
UIObject.invalidate()	Marks the object so it will be redrawn on the next frame interval.
UIObject.move()	Moves the object to the requested position.
UIObject.redraw()	Forces validation of the object so it is drawn in the current frame.
UIObject.setSize()	Resizes the object to the requested size.
UIObject.setSkin()	Sets a skin in the object.
UIObject.setStyle()	Sets the style property on the style declaration or object.

Methods inherited from the UIComponent class

The following table lists the methods the Window class inherits from the UIComponent class. When calling these methods from the Window object, use the form *WindowInstance.methodName*.

Method	Description
UIComponent.getFocus()	Returns a reference to the object that has focus.
UIComponent.setFocus()	Sets focus to the component instance.

Property summary for the Window class

The following table lists properties of the Window class.

Property	Description
Window.closeButton	Indicates whether a close button is (`true`) or is not (`false`) included on the title bar.
Window.content	A reference to the content (root movie clip) of the window.
Window.contentPath	Sets the name of the content to display in the window.
Window.title	The text that appears in the title bar.
Window.titleStyleDeclaration	The style declaration that formats the text in the title bar.

Properties inherited from the UIObject class

The following table lists the properties the Window class inherits from the UIObject class. When accessing these properties from the Window object, use the form *WindowInstance.propertyName*.

Property	Description
UIObject.bottom	The position of the bottom edge of the object, relative to the bottom edge of its parent. Read-only.
UIObject.height	The height of the object, in pixels. Read-only.
UIObject.left	The left edge of the object, in pixels. Read-only.
UIObject.right	The position of the right edge of the object, relative to the right edge of its parent. Read-only.
UIObject.scaleX	A number indicating the scaling factor in the *x* direction of the object, relative to its parent.
UIObject.scaleY	A number indicating the scaling factor in the *y* direction of the object, relative to its parent.
UIObject.top	The position of the top edge of the object, relative to its parent. Read-only.
UIObject.visible	A Boolean value indicating whether the object is visible (`true`) or not (`false`).
UIObject.width	The width of the object, in pixels. Read-only.
UIObject.x	The left edge of the object, in pixels. Read-only.
UIObject.y	The top edge of the object, in pixels. Read-only.

Properties inherited from the UIComponent class

The following table lists the properties the Window class inherits from the UIComponent class. When accessing these properties from the Window object, use the form *WindowInstance.propertyName*.

Property	Description
UIComponent.enabled	Indicates whether the component can receive focus and input.
UIComponent.tabIndex	A number indicating the tab order for a component in a document.

Event summary for the Window class

The following table lists the events of the Window class.

Event	Description
Window.click	Broadcast when the close button is clicked (released).

Event	Description
Window.complete	Broadcast when a window is created.
Window.mouseDownOutside	Broadcast when the mouse is clicked (released) outside the modal window.

Events inherited from the UIObject class

The following table lists the events the Window class inherits from the UIObject class.

Event	Description
UIObject.draw	Broadcast when an object is about to draw its graphics.
UIObject.hide	Broadcast when an object's state changes from visible to invisible.
UIObject.load	Broadcast when subobjects are being created.
UIObject.move	Broadcast when the object has moved.
UIObject.resize	Broadcast when an object has been resized.
UIObject.reveal	Broadcast when an object's state changes from invisible to visible.
UIObject.unload	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the Window class inherits from the UIComponent class.

Event	Description
UIComponent.focusIn	Broadcast when an object receives focus.
UIComponent.focusOut	Broadcast when an object loses focus.
UIComponent.keyDown	Broadcast when a key is pressed.
UIComponent.keyUp	Broadcast when a key is released.

Window.click

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(click){
   ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.click = function(eventObject){
  ...
}
windowInstance.addEventListener("click", listenerObject)
```

Description

Event; broadcast to all registered listeners when the mouse is clicked (released) over the close button.

The first usage example uses an `on()` handler and must be attached directly to a Window instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the Window instance `myWindow`, sends "_level0.myWindow" to the Output panel:

```
on(click){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*`windowInstance`*) dispatches an event (in this case, `click`) and the event is handled by a function, also called a *handler*, on a listener object (*`listenerObject`*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. The event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example creates a modal window and then defines a click handler that deletes the window. You must add a Window component to the Stage and then delete it to add the component to the document library; then add the following code to Frame 1:

```
import mx.managers.PopUpManager
import mx.containers.Window
var myTW = PopUpManager.createPopUp(_root, Window, true, {closeButton: true,
  title:"My Window"});
windowListener = new Object();
windowListener.click = function(evt){
  _root.myTW.deletePopUp();
}
myTW.addEventListener("click", windowListener);
```

See also

`EventDispatcher.addEventListener()`, `Window.closeButton`

Window.closeButton

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
windowInstance.closeButton
```

Description

Property; a Boolean value that indicates whether the title bar should have a close button (`true`) or not (`false`). This property must be set in the *`initObject`* parameter of the `PopUpManager.createPopUp()` method. The default value is `false`.

Clicking the close button broadcasts a `click` event, but doesn't close the window. You must write a handler that calls `Window.deletePopUp()` to explicitly close the window. For more information about the `click` event, see `Window.click`.

Example

The following code creates a window that displays the content in the movie clip "LoginForm" and has a close button on the title bar:

```
var myTW = PopUpManager.createPopUp(_root, Window, true,
  {contentPath:"LoginForm", closeButton:true});
```

See also

`PopUpManager.createPopUp()`, `Window.click`

Window.complete

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
listenerObject = new Object();
listenerObject.complete = function(eventObject){
  ...
}
windowInstance.addEventListener("complete", listenerObject)
```

Description

Event; broadcast to all registered listeners when a window is created. Use this event to size a window to fit its contents.

A component instance (*windowInstance*) dispatches an event (in this case, `complete`) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. The event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the `EventDispatcher.addEventListener()` method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example creates a window and then defines a `complete` handler that resizes the window to fit its contents. (This code would be placed on Frame 1 of the component in the library.)

```
import mx.managers.PopUpManager
import mx.containers.Window
var myTW = PopUpManager.createPopUp(_root, Window, true, {title:"Password
  Change", contentPath: "PasswordForm"});
lo = new Object();
lo.handleEvent = function(evtObj){
  if(evtObj.type == "complete"){
    _root.myTW.setSize(myTW.content._width, myTW.content._height + 25);
  }
}
myTW.addEventListener("complete", lo);
```

See also

`EventDispatcher.addEventListener()`

Window.content

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

windowInstance.content

Description

Property; a reference to the content (root movie clip) of the window. This property returns a MovieClip object. When you attach a symbol from the library, the default value is an instance of the attached symbol. When you load content from a URL, the default value is `undefined` until the load operation has started.

Example

The following code sets the value of the text property within the content inside the Window component:

```
myLoginWindow.content.password.text = "secret";
```

Window.contentPath

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
windowInstance.contentPath
```

Description

Property; sets the name of the content to display in the window. This value can be the linkage identifier of a movie clip in the library, or the absolute or relative URL of a SWF or JPEG file to load. The default value is `""` (an empty string).

Example

The following code creates a Window instance that displays the movie clip with the linkage identifier "LoginForm":

```
var myTW = PopUpManager.createPopUp(_root, Window, true,
  {contentPath:"LoginForm"});
```

Window.deletePopUp()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

```
windowInstance.deletePopUp()
```

Parameters

None.

Returns

Nothing.

Description

Method; deletes the window instance and removes the modal state. This method can be called only on Window instances that were created by `PopUpManager.createPopUp()`.

Example

The following code creates a modal window, then creates a listener that deletes the window when the close button is clicked:

```
import mx.managers.PopUpManager;
import mx.containers.Window;

var myTW:MovieClip = PopUpManager.createPopUp(_root, Window, true,
  {closeButton:true, title:"Test"});
var twListener:Object = new Object();
twListener.click = function(evt:Object){
   evt.target.deletePopUp();
}
myTW.addEventListener("click", twListener);
```

Note: Remember to add a Window component to the Stage then delete it to add it to the Library before running the above code.

Window.mouseDownOutside

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

Usage 1:

```
on(mouseDownOutside){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.mouseDownOutside = function(eventObject){
  ...
}
windowInstance.addEventListener("mouseDownOutside", listenerObject)
```

Description

Event; broadcast to all registered listeners when the mouse is clicked (released) outside the modal window. This event is rarely used, but you can use it to dismiss a window if the user tries to interact with something outside of it.

The first usage example uses an `on()` handler and must be attached directly to a Window instance. The keyword `this`, used inside an `on()` handler attached to a component, refers to the component instance. For example, the following code, attached to the Window instance `myWindowComponent`, sends "_level0.myWindowComponent" to the Output panel:

```
on(mouseDownOutside){
  trace(this);
}
```

The second usage example uses a dispatcher/listener event model. A component instance (*windowInstance*) dispatches an event (in this case, mouseDownOutside) and the event is handled by a function, also called a *handler*, on a listener object (*listenerObject*) that you create. You define a method with the same name as the event on the listener object; the method is called when the event is triggered. When the event is triggered, it automatically passes an event object (*eventObject*) to the listener object method. The event object has properties that contain information about the event. You can use these properties to write code that handles the event. Finally, you call the EventDispatcher.addEventListener() method on the component instance that broadcasts the event to register the listener with the instance. When the instance dispatches the event, the listener is called.

For more information, see "EventDispatcher class" on page 415.

Example

The following example creates a window instance and defines a mouseDownOutside handler that calls a beep() method if the user clicks outside the window:

```
var myTW = PopUpManager.createPopUp(_root, Window, true, undefined, true);
// create a listener
twListener = new Object();
twListener.mouseDownOutside = function()
{
    beep(); // make a noise if user clicks outside
}
myTW.addEventListener("mouseDownOutside", twListener);
```

See also

EventDispatcher.addEventListener()

Window.title

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

windowInstance.title

Description

Property; a string indicating the text of the title bar. The default value is "" (an empty string).

Example

The following code sets the title of the window to "Hello World":

```
myTW.title = "Hello World";
```

Window.titleStyleDeclaration

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX 2004.

Usage

windowInstance.titleStyleDeclaration

Description

Property; a string indicating the style declaration that formats the title bar of a window. The default value is `undefined`, which indicates bold, white text.

Example

With a Window component in the library, use the following ActionScript to format the style of the window's title bar.

```
// create a new CSSStyleDeclaration named TitleStyles
// and list it with the global styles list
_global.styles.TitleStyles = new mx.styles.CSSStyleDeclaration();
// customize styles
_global.styles.TitleStyles.fontStyle = "italic";
_global.styles.TitleStyles.textDecoration = "underline";
_global.styles.TitleStyles.color = 0xff0000;
_global.styles.TitleStyles.fontSize = 14;

tw = mx.managers.PopUpManager.createPopUp(this, mx.containers.Window, true,
  {closeButton:true, titleStyleDeclaration:"TitleStyles"});
tw.title = "Testing Styles";
tw.setSize(200, 100);
tw.move(20, 20);
var handleCloseObject:Object = new Object();
handleCloseObject.click = function(evt:Object) {
  evt.target.deletePopUp();
};
tw.addEventListener("click", handleCloseObject);
```

For more information about styles, see "Using styles to customize component color and text" on page 67.

XMLConnector component (Flash Professional only)

The XMLConnector component lets you read or write XML documents using HTTP `GET` or `POST` operations. It acts as a connector between other components and external XML data sources. The XMLConnector component communicates with components in your application using either ActionScript code or data binding features in the Flash authoring environment. The XMLConnector component has properties, methods, and events, but it has no visual appearance at runtime.

Using the XMLConnector component (Flash Professional only)

The XMLConnector component provides your application with access to any external data source that returns or receives XML through HTTP. The easiest way to connect with an external XML data source and use the parameters and results of that data source for your application is to specify a *schema,* the structure of the XML document that identifies the data elements in the document to which you can bind.

For more information on working with the XMLConnector component, see "Connecting to XML data with the XMLConnector component (Flash Professional only)" in *Using Flash*.

XMLConnector parameters

You can set the following authoring parameters for each XMLConnector component instance in the Parameters tab of the Component inspector:

direction is a string that defines what HTTP operation to perform when `trigger()` is called: `send`, `sendAndLoad`, or `load` correspond to `receive`, `receive/send`, and `send`, respectively.

ignoreWhite is a Boolean value; the default setting is `false`. When this parameter is set to `true`, the text nodes that contain only white space are discarded during the parsing process. Text nodes with leading or trailing white space are unaffected.

multipleSimultaneousAllowed is a Boolean value; when set to `true`, it allows a `trigger()` operation to initiate when another `trigger()` operation is already in progress. Multiple simultaneous `trigger()` operations may not be completed in the same order they were called. Also, Flash Player may place limits on the number of simultaneous network operations. This limit varies by version and platform. When the parameter is set to `false`, a `trigger()` operation cannot initiate if another one is in progress.

suppressInvalidCall is a Boolean value; when set to `true`, it suppresses the `trigger()` operation if the data parameters are invalid. When suppressInvalidCall is set to `false`, the `trigger()` operation executes and uses invalid data if necessary.

URL is a string that points to an external XML data source.

Common workflow for the XMLConnector component

The following procedure outlines the typical workflow for the XMLConnector component.

To use an XMLConnector component:

1. Add an instance of the XMLConnector component to your application and give it an instance name.
2. Use the Parameters tab of the Component inspector to enter the URL for the external XML data source that you want to access.
3. Use the Schema tab of the Component inspector to specify a schema for the XML document.

   Note: You can use the Import Sample Schema button to automate this process.

4. Use the Bindings tab of the Component inspector to bind data elements (`params` and `results`) from the XML document to properties of the visual components in your application.

   For example, you can connect to an XML document that provides weather data and bind the Location and Temperature data elements to label components in your application. The name and temperature of a specified city appears in the application at runtime.
5. Add a trigger to initiate the data binding operation by using one of the following methods:
   - Attach the Trigger Data Source behavior to a button.
   - Add your own ActionScript to call the `trigger()` method on the XMLConnector component.
   - Create a binding between an XML parameter and a UI control and set its Kind property to AutoTrigger. For more information, see "Schema kinds" in *Using Flash*.

For a step-by-step example that connects and displays XML using the XMLConnector component, see "XML Tutorial: Timesheet" in the Data Integration tutorials at www.macromedia.com/go/data_integration.

XMLConnector class (Flash Professional only)

Inheritance RPCCall > XMLConnector

ActionScript Class Name mx.data.components.XMLConnector

The XMLConnector class lets you send or receive XML files using HTTP. You can use ActionScript to bind other components to a data source that returns XML data, allowing communication between the components.

Method summary for the XMLConnector class

The following table lists the method of the XMLConnector class.

Method	Description
`XMLConnector.trigger()`	Initiates a remote procedure call.

Property summary for the XMLConnector class

The following table lists properties of the XMLConnector class.

Property	Description
XMLConnector.direction	Indicates whether data is being sent, received, or both.
XMLConnector.ignorewhite	Indicates whether text nodes containing only white space are discarded during the parsing process.
XMLConnector.multipleSimultaneousAllowed	Indicates whether multiple calls can take place at the same time.
XMLConnector.params	Specifies data that will be sent to the server when the next `trigger()` operation is executed.
XMLConnector.results	Identifies data that was received from the server as a result of the `trigger()` operation.
XMLConnector.suppressInvalidCalls	Indicates whether to suppress a call if parameters are invalid.
XMLConnector.URL	The URL used by the component in HTTP operations.

Event summary for the XMLConnector class

The following table lists events of the XMLConnector class.

Event	Description
XMLConnector.result	Broadcast when a remote procedure call completes successfully.
XMLConnector.send	Broadcast when the `trigger()` method is in process, after the parameter data has been gathered but before the data is validated and the remote procedure call is initiated.
XMLConnector.status	Broadcast when a remote procedure call is initiated, to inform the user of the status of the operation.

XMLConnector.direction

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.direction

Description

Property; indicates whether data is being sent, received, or both. The values are the following:

- `send` XML data for the `params` property is sent by HTTP `POST` method to the URL for the XML document. Any data that is returned is ignored. The `results` property is not set to anything, and there is no `result` event.

  Note: The `params` and `results` properties and the `result` event are inherited from the RPC component API.

- `receive` No `params` data is sent to the URL. The URL for the XML document is accessed through HTTP `GET`, and valid XML data is expected from the URL.
- `send/receive` The `params` data is sent to the URL, and valid XML data is expected from the URL.

Example

The following example sets the direction to `receive` for the document `mysettings.xml`:

```
myXMLConnector.direction = "receive";
myXMLConnector.URL = "mysettings.xml";
myXMLConnector.trigger();
```

XMLConnector.ignorewhite

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

`componentInstance``.ignoreWhite`

Description

Property; a Boolean value. When this parameter is set to `true`, the text nodes that contain only white space are discarded during the parsing process. Text nodes with leading or trailing white space are unaffected. The default setting is `false`.

Example

The following code sets the `ignoreWhite` property to `true`:

```
myXMLConnector.ignoreWhite = true;
```

XMLConnector.multipleSimultaneousAllowed

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.multipleSimultaneousAllowed

Description

Property; indicates whether multiple calls can take place at the same time. If this property is `false`, the `XMLConnector.trigger()` method performs a call if another call is already in progress. A `status` event is emitted, with the code `CallAlreadyInProgress`. If this property is `true`, the call takes place.

When multiple calls are simultaneously in progress, there is no guarantee that they will complete in the order in which they were triggered. Also, the browser and/or operating system may place limits on the number of simultaneous network operations. The most likely limit you may encounter is the browser enforcing a maximum number of URLs that can be downloaded simultaneously. This is something that is often configurable in a browser. However, even in this case, the browser should queue streams, and this should not interfere with the expected behavior of the Flash application.

Example

This example retrieves a remote XML file using the XMLConnector component by setting the `direction` property to `receive`. Drag an XMLConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.XMLConnector;
var xmlListener:Object = new Object();
xmlListener.result = function(evt:Object) {
  trace("results:");
  trace(evt.target.results);
  trace("");
};
xmlListener.status = function(evt:Object) {
  trace("status::"+evt.code);
};
var myXMLConnector:XMLConnector = new XMLConnector();
myXMLConnector.addEventListener("result", xmlListener);
myXMLConnector.addEventListener("status", xmlListener);
myXMLConnector.direction = "receive";
myXMLConnector.URL = "http://www.flash-mx.com/mm/tips/tips.xml";
myXMLConnector.multipleSimultaneousAllowed = false;
myXMLConnector.suppressInvalidCalls = true;
myXMLConnector.trigger();
myXMLConnector.trigger();
myXMLConnector.trigger();
```

This example specifies the URL of the XML file, and sets `multipleSimultaneousAllowed` to `false`. It triggers the XMLConnector instance three times, which causes the event listener's `status` method to display the error code `CallAlreadyInProgress` two times in the Output panel. The first attempt is successfully sent from Flash to the server. When the first trigger successfully receives a result, the `result` event is broadcast and the XML packet you receive is displayed in the Output panel.

XMLConnector.params

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.params

Description

Property; specifies data that will be sent to the server when the next `trigger()` operation is executed. Each RPC component defines how this data is used, and what the valid types are.

Example

The following example defines `name` and `city` parameters for `myXMLConnector`:

```
myXMLConnector.params = new XML("<mydoc><name>Bob</name><city>Oakland</city></
  mydoc>");
```

XMLConnector.result

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.addEventListener("result", *myListenerObject*)

Description

Event; broadcast when a remote procedure call completes successfully.

The parameter to the event handler is an object with the following fields:

- `type`: the string `"result"`
- `target`: a reference to the object that emitted the event (for example, a WebServiceConnector component)

You can retrieve the actual result value using the `results` property.

Example

The following example defines a function `res` for the `result` event and assigns the function to the `addEventListener` event handler:

```
var res = function (ev) {
trace(ev.target.results);
};
xcon.addEventListener("result", res);
```

XMLConnector.results

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.results

Description

Property; identifies data that was received from the server as a result of a `trigger()` operation. Each RPC component defines how this data is fetched, and what the valid types are. This data appears when the RPC operation has successfully completed, as signaled by the `result` event. It is available until the component is unloaded, or until the next RPC operation.

It is possible for the returned data to be very large. You can manage this in two ways:

- Select an appropriate movie clip, Timeline, or screen as the parent for the RPC component. The component's memory will become available for garbage collection when the parent is destroyed.
- In ActionScript, you can assign `null` to this property at any time.

Example

The following example traces the `results` property for `myXMLConnector`:

```
trace(myXMLConnector.results);
```

XMLConnector.send

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.addEventListener("send", *myListenerObject*)

Description

Event; broadcast when the `trigger()` operation is in process, after the parameter data has been gathered but before the data is validated and the remote procedure call is initiated. This is a good place to put code that will modify the parameter data before the call.

The parameter to the event handler is an object with the following fields:

- `type`: the string `"send"`
- `target`: a reference to the object that emitted the event (for example, a WebServiceConnector component)

You can retrieve or modify the actual parameter values by using the `params` property.

Example

The following example defines a function `sendFunction` for the `send` event and assigns the function to the `addEventListener` event handler:

```
var sendFunction = function (sendEnv) {
sendEnv.target.params = [newParam_txt.text];
};
xcon.addEventListener("send", sendFunction);
```

XMLConnector.status

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.addEventListener("status", *myListenerObject*)

Description

Event; broadcast when a remote procedure call is initiated, to inform the user of the status of the operation.

The parameter to the event handler is an object with the following fields:

- `type`: the string `"status"`
- `target`: a reference to the object that emitted the event (for example, a WebServiceConnector component)
- `code`: a string giving the name of the specific condition that occurred
- `data`: an object whose contents depend on the code

The code field for the status event is set to Fault if problems occur with the call, as follows:

Code	Data	Description
Fault	{faultcode: code, faultstring: string, detail: detail, element: element, faultactor: actor}	This event is emitted if other problems occur during the processing of the call. The data is a SOAPFault object. After this event occurs, the attempted call is considered complete, and there is no `result` or `send` event.

The following are the faults that can occur with the `status` event:

FaultCode	FaultString	Notes
`XMLConnector.Not.XML`	params is not an XML object	The `params` value must be an ActionScript XML object.
`XMLConnector.Parse.Error`	params had XML parsing error NN.	The `status` property of the `params` XML object had a nonzero value NN. To see the possible errors NN, see `XML.status` in *Flash ActionScript Language Reference*.
`XMLConnector.No.Data.Received`	no data was received from the server	Due to various browser limitations, this message can mean either (a) the server URL was invalid, did not respond, or returned an HTTP error code; or (b) the server request succeeded but the response was 0 bytes of data. To work around this restriction, design your application so that the server never returns 0 bytes of data. If you receive the fault code `XMLConnector.No.Data.Received`, you will know that there was a server error, and can inform the user accordingly.
`XMLConnector.Results.Parse.Error`	received data had an XML parsing error NN	The received XML was not valid, as determined by the Flash Player built-in XML parser. To see the possible errors NN, see `XML.status` in *Flash ActionScript Language Reference*.
`XMLConnector.Params.Missing`	Direction is 'send' or 'send/receive', but params are null.	

Example

The following example defines a function `statusFunction` for the `status` event and assigns the function to the `addEventListener` event handler:

```
var statusFunction = function (stat) {
trace(stat.code);
trace(stat.data.faultcode);
trace(stat.data.faultstring);
};
xcon.addEventListener("status", statusFunction);
```

XMLConnector.suppressInvalidCalls

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.suppressInvalidCalls

Description

Property; indicates whether to suppress a call if parameters are invalid. If this property is `true`, the `trigger()` method does not perform a call if the bound parameters fail the validation. A `status` event is emitted, with the code `InvalidParams`. If this property is `false`, the call takes place, using the invalid data as required.

Example

This example displays an error because the required parameters are not being passed. Drag an XMLConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.XMLConnector;
var xmlListener:Object = new Object();
xmlListener.result = function(evt:Object) {
  trace("results:");
  trace(evt.target.results);
  trace("");
};
xmlListener.status = function(evt:Object) {
  switch (evt.code) {
  case 'Fault' :
    trace("ERROR! ["+evt.data.faultcode+"]");
    trace("\t"+evt.data.faultstring);
    break;
  case 'InvalidParams' :
    trace("ERROR! ["+evt.code+"]");
    break;
  }
};
var myXMLConnector:XMLConnector = new XMLConnector();
myXMLConnector.addEventListener("result", xmlListener);
myXMLConnector.addEventListener("status", xmlListener);
myXMLConnector.direction = "send/receive";
myXMLConnector.URL = "http://www.flash-mx.com/mm/login_xml.cfm";
myXMLConnector.multipleSimultaneousAllowed = false;
myXMLConnector.suppressInvalidCalls = false;
// myXMLConnector.params = new XML("<login username='Mort'
  password='Guacamole' />");
myXMLConnector.trigger();
```

Remove the comments from the second to last line of code for the snippet to work correctly.

XMLConnector.trigger()

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.trigger()

Description

Method; initiates a remote procedure call. Each RPC component defines exactly what this involves. If the operation is successful, the results of the operation appear in the RPC component's `results` property.

The `trigger()` method performs the following steps:

1. If any data is bound to the `params` property, the method executes all the bindings to ensure that up-to-date data is available. This also causes data validation to occur.
2. If the data is not valid and `suppressInvalidCalls` is set to `true`, the operation is discontinued.
3. If the operation continues, the `send` event is emitted.
4. The actual remote call is initiated using the connection method indicated (for example, HTTP).

Example

This example retrieves a remote XML file using the XMLConnector by setting the `direction` property to `receive`. Drag an XMLConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.XMLConnector;
var xmlListener:Object = new Object();
xmlListener.result = function(evt:Object) {
  trace("results:");
  trace(evt.target.results);
  trace("");
};
xmlListener.status = function(evt:Object) {
  trace("status::"+evt.code);
};
var myXMLConnector:XMLConnector = new XMLConnector();
myXMLConnector.addEventListener("result", xmlListener);
myXMLConnector.addEventListener("status", xmlListener);
myXMLConnector.direction = "receive";
myXMLConnector.URL = "http://www.flash-mx.com/mm/tips/tips.xml";
myXMLConnector.multipleSimultaneousAllowed = false;
myXMLConnector.suppressInvalidCalls = true;
myXMLConnector.trigger();
myXMLConnector.trigger();
myXMLConnector.trigger();
```

This code specifies the URL of the XML file and sets `multipleSimultaneousAllowed` to `false`. It triggers the XMLConnector instance three times, which causes the event listener's `status` method to display the error code `CallAlreadyInProgress` two times in the Output panel. The first attempt is successfully sent from Flash to the server. When the first trigger successfully receives a result, the `result` event is broadcast and the XML packet you receive is displayed in the Output panel.

XMLConnector.URL

Availability

Flash Player 6 (6.0 79.0).

Edition

Flash MX Professional 2004.

Usage

componentInstance.URL

Description

Property; the URL that this component uses when carrying out HTTP operations. This URL may be an absolute or relative URL. The URL is subject to all the standard Flash Player security protections.

Example

This example retrieves a remote XML file using the XMLConnector component by setting the `direction` property to `receive`. Drag an XMLConnector component into your library, and enter the following code on Frame 1 of the Timeline:

```
import mx.data.components.XMLConnector;
var xmlListener:Object = new Object();
xmlListener.result = function(evt:Object) {
  trace("results:");
  trace(evt.target.results);
  trace("");
};
xmlListener.status = function(evt:Object) {
  trace("status::"+evt.code);
};
var myXMLConnector:XMLConnector = new XMLConnector();
myXMLConnector.addEventListener("result", xmlListener);
myXMLConnector.addEventListener("status", xmlListener);
myXMLConnector.direction = "receive";
myXMLConnector.URL = "http://www.flash-mx.com/mm/tips/tips.xml";
myXMLConnector.multipleSimultaneousAllowed = false;
myXMLConnector.suppressInvalidCalls = true;
myXMLConnector.trigger();
myXMLConnector.trigger();
myXMLConnector.trigger();
```

This code specifies the URL of the XML file and sets `multipleSimultaneousAllowed` to `false`. It triggers the XMLConnector instance three times, which causes the event listener's `status()` method to display the error code `CallAlreadyInProgress` two times in the Output panel. The first attempt is successfully sent from Flash to the server. When the first trigger successfully receives a result, the `result` event is broadcast and the XML packet you receive is displayed in the Output panel.

XUpdateResolver component (Flash Professional only)

Resolver components are used with the DataSet component (part of the data management functionality in the Flash data architecture) to save changes to an external data source. Resolvers take a `DataSet.deltaPacket` object and convert it to an update packet in a format appropriate to the type of resolver. The update packet can then be transmitted to the external data source by one of the connector components. Resolver components have no visual appearance at runtime.

For general information on how to manage data in Flash using the DataSet component, see "Data management (Flash Professional only)" in *Using Flash*.

XUpdate is a standard for describing changes that are made to an XML document and is supported by a variety of XML databases, such as Xindice and XHive. The XUpdateResolver component translates the changes made to a DataSet component into XUpdate statements. The updates from the XUpdateResolver component are sent in the form of an XUpdate data packet, which is communicated to the database or server through a connection object. The XUpdateResolver component gets a delta packet from a DataSet component, sends its own update packet to a connector, receives server errors back from the connection, and communicates them back to the DataSet component—all using bindable properties.

For information about the working draft of the XUpdate language specification, see www.xmldb.org/xupdate/xupdate-wd.html. For more information, also see "RDBMSResolver component (Flash Professional only)" on page 636, "DataSet component (Flash Professional only)" on page 301, "WebServiceConnector component (Flash Professional only)" on page 865, and "XMLConnector component (Flash Professional only)" on page 894. For information about the Flash data architecture, see "Data resolution (Flash Professional only)" in *Using Flash*; for information about resolving XML data, see "Resolving XML data with the XUpdateResolver component (Flash Professional only)" in *Using Flash*.

Note: You can also use the XUpdateResolver component to send data updates to any external data source that can parse the XUpdate language—for example, an ASP page, a Java servlet, or a ColdFusion component.

Using the XUpdateResolver component (Flash Professional only)

The XUpdateResolver component is used only when your Flash application contains a DataSet component and must send an update back to an external data source.

The XUpdateResolver component communicates with the DataSet component by using the DataSetDeltaToXUpdateDelta encoder. This encoder creates XPath statements that uniquely identify nodes within an XML file according to the information contained in the DataSet component's delta packet. This information is used by the XUpdateResolver component to generate XUpdate statements. For more information on the DataSetDeltaToXUpdateDelta encoder, see "Schema encoders" in *Using Flash*.

For more information on working with the XUpdateResolver component, see "Data resolution (Flash Professional only)" in *Using Flash*.

XUpdateResolver component parameter

The XUpdateResolver component has one authoring parameter, the Boolean `includeDeltaPacketInfo` parameter. When this parameter is set to `true`, the update packet includes additional information that can be used by an external data source to generate results that can be sent back to your application. This information includes a unique transaction and operation ID that is used internally by the data set.

Note: The additional information that is included in the update packet invalidates the XUpdate. You would choose to add this information only if you were going to store it in a server object and use it to generate a result packet. In this scenario, your server object would pull the information out of the update packet for it s own needs and then pass on the (now valid) XUpdate to the database.

The following is an example of an XML update packet when the `includeDeltaPacketInfo` parameter is set to `false`:

```
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/
  xupdate">
    <xupdate:remove select="/datapacket/row[@id='100']"/>
</xupdate:modifications>
```

The following is an example of an XML update packet when the `includeDeltaPacketInfo` parameter is set to `true`:

```
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/
  xupdate"
      transId="46386292065:Wed Jun 25 15:52:34 GMT-0700 2003">
    <xupdate:remove select="/datapacket/row[@id='100']" opId="0123456789"/>
</xupdate:modifications>
```

Common workflow for the XUpdateResolver component

The following procedure outlines the typical workflow for the XUpdateResolver component.

To use an XUpdateResolver component:

1. Add two instances of the XMLConnector component and one instance each of the DataSet component and the XUpdateResolver component to your application, and give them instance names.
2. Select the first XMLConnector component, and use the Parameters tab of the Component inspector to enter the URL for the external XML data source that you want to access.
3. With the XMLConnector component still selected, click the Schema tab of the Component inspector and import a sample XML file to generate your schema.

   Note: You may need to create a virtual schema for your XML file if you want to access a subelement of the array that you are binding to the data set. For more information, see "Virtual schemas" in *Using Flash*.
4. Use the Bindings tab of the Component inspector to bind an array in the XMLConnector component to the `dataProvider` property of the DataSet component.
5. Select the DataSet component and use the Schema tab of the Component inspector to create the DataSet fields that will be bound to the fields of the object within the array.

6. Use the Bindings tab of the Component inspector to bind data elements (DataSet fields) to the visual components in your application.
7. Select the Schema tab of the XUpdateResolver component. With the `deltaPacket` component property selected, use the Schema Attributes pane to set the `encoder` property to the DataSetDeltaToXUpdateDelta encoder.
8. Select Encoder Options and enter the `rowNodeKey` value that uniquely identifies the row node within the XML file.

   Note: The `rowNodeKey` value combines an XPath statement with a field parameter to define how unique XPath statements should be generated for the update data contained within the delta packet. See information on the DataSetDeltaToXUpdateDelta encoder in "Schema encoders" in *Using Flash*.

9. Click the Bindings tab and create a binding between the XUpdateResolver component's `deltaPacket` property and the DataSet component's `deltaPacket` property.
10. Create another binding from the `xupdatePacket` property to the second XMLConnector component to send the data back to the external data source.

    Note: The `xupdatePacket` property contains the formatted delta packet (XUpdate statements) that will be sent to the server.

11. Add a trigger to initiate the data binding operation: use the Trigger Data Source behavior attached to a button, or add ActionScript.

In addition to these steps, you can also create bindings to apply the result packet sent back from the server to the data set by the XUpdateResolver component.

For a step-by-step example that resolves data to an external data source using XUpdate, see "Update the timesheet" in the Data Integration tutorials at www.macromedia.com/go/data_integration.

XUpdateResolver class (Flash Professional only)

Inheritance MovieClip > XUpdateResolver

ActionScript Class Name mx.data.components.XUpdateResolver

The properties and events of the XUpdateResolver class allow you to work with the DataSet component to save changes to external data sources.

Property summary for the XUpdateResolver class

The following table lists properties of the XUpdateResolver class.

Property	Description
`XUpdateResolver.deltaPacket`	Contains a description of the changes to the DataSet component. The DataSet component's `deltaPacket` property should be bound to this property so that when `DataSet.applyUpdates()` is called, the binding will copy it across and the resolver will create the XUpdate packet.
`XUpdateResolver.includeDeltaPacketInfo`	Includes additional information from the delta packet in attributes on the XUpdate nodes.
`XUpdateResolver.updateResults`	Describes results of an update.
`XUpdateResolver.xupdatePacket`	Contains the XUpdate translation of the changes to the DataSet component.

Event summary for the XUpdateResolver class

The following table lists events of the XUpdateResolver class.

Event	Description
`XUpdateResolver.beforeApplyUpdates`	Called by the resolver component to make custom modifications immediately after the XML packet has been created and immediately before that packet is sent.
`XUpdateResolver.reconcileResults`	Called by the resolver component to compare two packets.

XUpdateResolver.beforeApplyUpdates

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
resolveData.beforeApplyUpdates(eventObject)
```

Parameters

eventObject Resolver event object; describes the customizations to the XML packet before the update is sent through the connector to the database. This event object should contain the following properties:

Property	Description
target	Object; the resolver generating this event.
type	String; the name of the event.
updatePacket	XML object; the XML object that is about to be applied.

Returns

None.

Description

Event; called by the resolver component to make custom modifications immediately after the XML packet has been created for a new delta packet, and immediately before that packet is sent out using data binding. You can use this event handler to make custom modifications to the XML before sending the updated data to a connector.

Example

The following example adds the user authentication data to the XML packet:

```
on (beforeApplyUpdates) {
   // add user authentication data
   var userInfo = new XML(""+getUserId()+" "+getPassword()+"");
   xupdatePacket.firstChild.appendChild(userInfo);
}
```

XUpdateResolver.deltaPacket

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.deltaPacket

Description

Property; contains a description of the changes to the DataSet component. This property is of type deltaPacket and receives a delta packet to be translated into an XUpdate packet, and outputs a delta packet from any server results placed in the updateResults property. This property provides a way for you to make custom modifications to the XML before sending the updated data to a connector.

Messages in the `updateResults` property are treated as errors. This means that a delta with messages is added to the delta packet again so it can be re-sent the next time the delta packet is sent to the server. You must write code that handles deltas that have messages so that the messages are presented to the user and the deltas can be modified before being added to the next delta packet.

The DataSet component's `deltaPacket` property should be bound to this property so that when `DataSet.applyUpdates()` is called, the binding will copy it across and the resolver will create the XUpdate packet.

XUpdateResolver.includeDeltaPacketInfo

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.includeDeltaPacketInfo`

Description

Property; a Boolean property that, if `true`, includes additional information from the delta packet in attributes on the XUpdate nodes. This information consists of the transaction ID and operation ID.

For an example of the resulting XML, see "XUpdateResolver component parameter" on page 908.

XUpdateResolver.reconcileResults

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.reconcileResults(`*`eventObject`*`)`

Parameters

`eventObject` ResolverEvent object; describes the event object used to compare two update packets. This event object should contain the following properties:

Property	Description
`target`	Object; the resolver generating this event.
`type`	String; the name of the event.

Returns

None.

Description

Event; called by the resolver component to compare two packets. Use this callback to insert any code after the results have been received from the server and immediately before the transmission, through data binding, of the delta packet that contains operation results. This is a good place to put code that handles messages from the server.

Example

The following example reconciles two update packets and clears the updates on success:

```
on (reconcileResults) {
      // examine results
      if(examine(updateResults))
        myDataSet.purgeUpdates();
      else
       displayErrors(results);
}
```

XUpdateResolver.updateResults

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.updateResults

Description

Property; property of type `deltaPacket` that contains the results of an update returned from the server using a connector. Use this property to transmit errors and updated data from the server to a DataSet component—for example, when the server assigns new IDs for an auto-assigned field. Bind this property to a connector's `results` property so that it can receive the results of an update and transmit the results back to the DataSet component.

Messages in the `updateResults` property are treated as errors. This means that a delta with messages is added to the delta packet again so it can be re-sent the next time the delta packet is sent to the server. You must write code that handles deltas that have messages so that the messages are presented to the user and the deltas can be modified before being added to the next delta packet.

XUpdateResolver.xupdatePacket

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
resolveData.xupdatePacket
```

Description

Property; property of type `xml` that contains the XUpdate translation of the changes to the DataSet component. Bind this property to the connector component's property that transmits the translated update packet of changes back to the DataSet component.

CHAPTER 7
Creating Components

This chapter describes how to create your own component and package it for deployment.

The version 2 architecture has simplified the component development process in several ways. For more information, see "What's new in version 2 components" on page 916.

This chapter contains the following sections:

Component source files

The source files for the version 2 components that install with Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 are also installed with Flash. It is helpful to open a few of these files, look through them, and try to understand their structure before you build your own components. All the components are symbols in the library of StandardComponents.fla. Each symbol is linked to an ActionScript class, as follows:

- FLA file source code: First Run/ComponentFLA/StandardComponents.fla
- ActionScript class files: First Run/Classes/mx

What's new in version 2 components

The current version (version 2) of the Macromedia Component Architecture is very different from the Macromedia Flash MX version (version 1). The following list provides an overview of some of the changes that affect component developers:

New base classes provide a new component hierarchy. In version 1 components, the base class was FUIComponent. In version 2, the base classes are the UIObject class and UIComponent class. See "Selecting a parent class" on page 924.

Manager classes are system-level static classes that provide common, reusable pieces of functionality. The version 2 architecture includes a Focus Manager (mx.managers.FocusManager), Popup Manager (mx.managers.PopUpManager), Style Manager (mx.styles.StyleManager), and a Depth Manager (mx.managers.DepthManager).

The built-in live preview allows users to see changes to a component while authoring. In Flash MX, you had to create a live-preview SWF file for each component. In Flash MX 2004, Macromedia has made this much easier by allowing you to compile your component into an SWC file or compiled clip that automatically contains a live-preview file. See "Exporting and distributing a component" on page 953.

Metadata tag attributes allow you to inform the Flash Integrated Development Environment (IDE) and compiler about the parameters and attributes of your component. For example, you can specify component parameters in the component's class file rather than in the Component Definition dialog box. See "Adding component metadata" on page 935.

The broadcaster/listener event model allows components to send events to more than one listener. See "Dispatching events" on page 948.

CSS-based styles provide a powerful, standards-based method for configuring the look and feel of components. See "About styles" on page 950.

ActionScript 2.0 class files contain all the component code. In Flash MX, you had to have all your code inside the FLA file in an `#initclip #endinitclip` block. When creating version 2 components, you write your code in ActionScript 2.0 in an external class file and specify the component's class in the Linkage Properties and Component Definition dialog boxes. See "Creating the ActionScript class file" on page 930.

The SWC file format allows you to export your component in a package and prevents you from needing to recompile code within those packages. See "Exporting and distributing a component" on page 953.

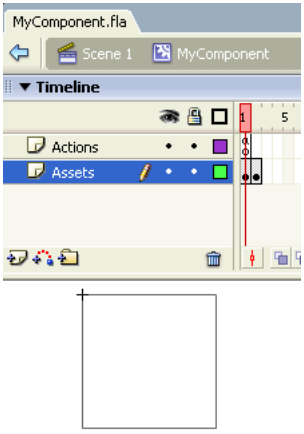
Overview of component structure

A component is comprised of a Flash (FLA) file and an ActionScript (AS) file. There are other files (for example, an icon and a .swd debugging file) that you can optionally create and package with your component, but all components require a FLA and an ActionScript file. When you've finished developing your component, you export it as a SWC file.

A Flash (FLA) file, an ActionScript (AS) file, and a SWC file

The FLA file contains a movie clip symbol that must be linked to the AS file in both the Linkage Properties and the Component Definition dialog boxes.

The movie clip symbol has two-frames and two-layers. The first layer is an Actions layer and has a `stop()` action on Frame 1. The second layer is an Assets layer with two keyframes. Frame 1 contains a bounding box or any graphics that serve as placeholders for the final art. Frame 2 contains all other assets, including graphics and base classes, used by the component.



The ActionScript code specifies the properties and methods for the component, and specifies which, if any, classes the component extends. The name of the AS file is the name of the component. For example, MyComponent.as contains the source code for the MyComponent component.

It's a good idea to save the component's FLA and AS files in the same folder and give them the same name. If the AS file is not saved in the same folder, you must verify that the folder is in the classpath so the FLA file can find it. For more information about the classpath, see "Understanding the classpath" in *Using ActionScript in Flash*.

Building your first component

In this section, you will build a Dial component. The completed component files, Dial.fla, Dial.as, Dial.swf, and DialAssets.fla are located in the examples folder on your hard disk:

- Macromedia/Flash MX 2004/Samples/HelpExamples/DialComponent

The Dial component is a potentiometer. A user can click on the needle and drag it to change its position. The API for the Dial has one property, `value`, that you can use to get and set the position of the needle.

- "Creating the Dial Flash (FLA) file" on page 918
- "Creating the Dial class file" on page 920
- "Testing and exporting the Dial component" on page 922

This section is much like a tutorial. These same procedures are discussed in more detail in subsequent sections. (See "Selecting a parent class" on page 924.)

Creating the Dial Flash (FLA) file

First you must create a Flash (FLA) file.

To create the Dial FLA file:

1. In Flash, choose File > New and create a new document.
2. Choose File > Save As and save the file as **Dial.fla**.

   The file can have any name, but giving it the same name as the component is practical.
3. Choose Insert > New Symbol.

   Give it the name **Dial**, and the behavior Movie clip.
4. If the Linkage section of the Create New Symbol dialog isn't open, click the Advanced button to reveal it.
5. Select Export for ActionScript and deselect Export in First Frame.
6. Enter the AS 2.0 Class **Dial**.

   This value is the component class name. If the class is in a package (for example, mx.controls.Button), enter the entire package name.
7. Click OK.

   Flash opens into Edit Symbol mode.
8. Insert a new layer.

   Name the top layer **Actions** and the bottom layer **Assets**.
9. Select Frame 2 in the Assets layer and insert a keyframe (F6).

   This is the structure of the component movie clip: an Actions layer and an Assets layer. The Actions layer has 1 keyframe and the Assets layer has 2 keyframes.
10. Select Frame 1 in the Actions layer and open the Actions panel (F9). Enter a `stop();` action.

    This prevents the movie clip from proceeding to Frame 2.

11. Choose File > Import > Open External Library and choose the StandardComponents.fla file from the FirstRun/ComponentFLA folder.

    Dial extends the UIComponent base class; therefore, you must drag an instance of the UIComponent class into the Dial document.

    Note: For information about folder locations, see "Configuration folders installed with Flash" in *Using Flash*.

12. In the StandardComponents library, browse to the UIComponent movie clip in the following folders: Flash UI Components 2 > Base Classes > FUIObject Subclasses and drag it to the Dial.fla library.

    Other assets are copied to the Dial library with UIComponent.

    Note: When you drag UIComponent to the Dial library, the folder hierarchy in the library is changed. If you plan to use your library again, or use it with other groups of components (such as the version 2 components), you should restructure the folder hierarchy to match the StandardComponents library so that it's organized well and you avoid duplicate symbols.

13. Close the StandardComponents library.
14. In the Assets layer, select Frame 2 and drag an instance of UIComponent to the Stage.
15. Choose File > Import > Open External Library and choose the DialAssets.fla file from the Macromedia/Flash MX 2004/Samples/HelpExamples/DialComponent folder.
16. In the Assets layer, select Frame 2 and drag an instance of the DialFinal movie clip from the DialAssets library to the Dial library.

    All the component assets are added to Frame 2 of the Assets layer. Because there is a `stop()` action on Frame 1 of the Actions layer, the assets in Frame 2 won't be seen as they are arranged on the Stage. You add them to Frame 2 so that they exist in the component library and you can instantiate them dynamically (in the case of DialFinal), or access their methods, properties, and events (in the case of UIComponent).

17. In the Assets layer, select Frame 1. Drag the BoundingBox movie clip from the library (Flash UI Components 2 > Component Assets folder) to the Stage. Give the BoundingBox the instance name **boundingBox_mc**. Use the Info panel to resize it to the size of the DialFinal movie clip (250, 250), and position it at 0, 0.

    The BoundingBox instance is used to create the component's live preview and handle resizing during authoring. You must size the bounding box so that it can enclose all the graphical elements in your component.

    Note: If you are extending a component (including any version 2 component) that uses the instance name `boundingBox_mc`, you must use that name too. Otherwise, you can use any instance name.

18. Select the Dial movie clip in the library, and choose Component Definition from the Library options menu.
19. In the AS 2.0 Class text box, enter **Dial**.

    This value is the name of the ActionScript class. If the class is in a package, the value is the full package, for example, mx.controls.CheckBox.

20. Save the file.

Creating the Dial class file

The following is the ActionScript class for the Dial component. Please read the comments in the code for a description of each section.

To create this file, you can create a new ActionScript file in the Flash IDE, or use any other text editor. Save the file as Dial.as in the same folder as the Dial.fla file.

```
// Import the package so you can reference
// the class directly.
import mx.core.UIComponent;

// Event metadata tag
[Event("change")]
class Dial extends UIComponent
{
  // Components must declare these to be proper
  // components in the components framework.
  static var symbolName:String = "Dial";
  static var symbolOwner:Object = Dial;
  var className:String = "Dial";

  // The needle and dial movie clips that are
  // the component's graphical representation
  private var needle:MovieClip;
  private var dial:MovieClip;
  private var boundingBox_mc:MovieClip;

  // The "value" property is a getter/setter,
  // so that you can update the needle's position
  // when the value is set. This is a private
  // variable that stores the value.
  private var __value:Number = 0;

  // This flag is set when the user drags the
  // needle with the mouse, and cleared afterwards.
  private var dragging:Boolean = false;

  // Constructor; does nothing,
  // but is required for all classes.
  function Dial() {
  }

  // Initialization code
  function init():Void {
    super.init();
    useHandCursor = false;
    boundingBox_mc._visible = false;
    boundingBox_mc._width = 0;
    boundingBox_mc._height = 0;
  }

  private function createChildren():Void
  {
    dial = createObject("DialFinal", "dial", 10);
```

```
    size();
}

// The draw method is invoked after the component has
// been invalidated, by someone calling invalidate().
// This is better than redrawing from within the set function
// for value, because if there are other properties, it's
// better to batch up the changes into one redraw, rather
// than doing them all individually. This approach leads
// to more efficiency and better centralization of code.
function draw():Void {
    super.draw();
    dial.needle._rotation = value;
}

// The size method is invoked when the component's size
// changes. This is an opportunity to resize the children,
// and the dial and needle graphics.
function size():Void {
    super.size();
    dial._width  = width;
    dial._height = height;
    // Cause the needle to get redrawn, in case it needs it
    invalidate();
}

// This is the getter/setter for the value property.
// The [Inspectable] metadata makes the property appear
// in the Property inspector. This is a getter/setter
// so that you can call invalidate and force the component
// to redraw, when the value is changed.
[Bindable]
[ChangeEvent("change")]
[Inspectable(defaultValue=0)]
function set value (val:Number)
{
    __value = val;
    invalidate();
}

function get value ():Number
{
    return twoDigits(__value);
}

function twoDigits(x:Number):Number
{
    return (Math.round(x * 100) / 100);
}


// Tells the component to expect mouse presses
function onPress()
{
    beginDrag();
```

```
    }

    // When the dial is pressed, the dragging flag is set.
    // The mouse events are assigned callback functions.
    function beginDrag()
    {
      dragging = true;
      onMouseMove = mouseMoveHandler;
      onMouseUp = mouseUpHandler;
    }

    // Remove the mouse events when the drag is complete,
    // and clear the flag.
    function mouseUpHandler()
    {
      dragging = false;
      delete onMouseMove;
      delete onMouseUp;
    }

    function mouseMoveHandler()
    {
      // Figure out the angle
      if (dragging) {
        var x:Number = _xmouse - width/2;
        var y:Number = _ymouse - height/2;

        var oldValue:Number = value;
        var newValue:Number = 90+180/Math.PI*Math.atan2(y, x);
        if (newValue<0) {
          newValue += 360;
        }
        if (oldValue != newValue) {
          value = newValue;
          dispatchEvent( {type:"change"} );
        }
      }
    }
  }
```

Testing and exporting the Dial component

You've created the Flash file that contains the graphical elements and the base classes and the class file that contains all the functionality of the Dial component. Now it's time to test the component.

Ideally, you would test the component as you work, especially while you're writing the class file. The fastest way to test as you work is to convert the component to a compiled clip and use it in the component's FLA file.

When you've completely finished a component, export it as a SWC file. For more information, see "Exporting and distributing a component" on page 953.

To test the Dial component:

1. In the Dial.fla file, choose the Dial component in the library and choose Convert to Compiled Clip from the Library options menu.

   A compiled clip is added to the library with the name **Dial SWF**.

   Note: If you've already created a compiled clip (for example, if this is the second or third time you're testing), a Resolve Library Conflict dialog box appears. Choose Replace Existing Items to add the new version to the document.

2. Drag Dial SWF to the Stage on the main Timeline.
3. You can resize it and set its value property in the Property inspector or the Component Inspector. When you set its value property, the needle's position should change accordingly.
4. To test the `value` property at runtime, give the dial the instance name **dial** and add the following code to Frame 1 on the main Timeline:

```
// position of the text field
var textXPos:Number = dial.width/2 + dial.x
var textYPos:Number = dial.height/2 + dial.y;

// creates a text field in which to view the dial.value
createTextField("dialValue", 10, textXPos, textYPos, 100, 20);

// creates a listener to handle the change event
function change(evt){
// places the value property in the text field
// whenever the needle moves
  dialValue.text = dial.value;
}
dial.addEventListener("change", this);
```

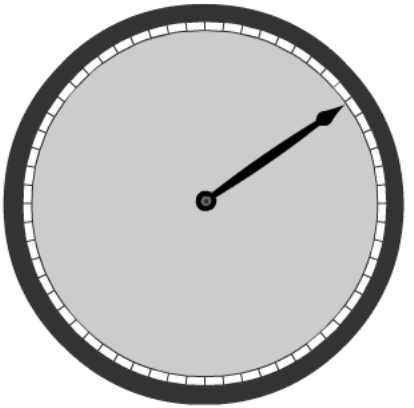
5. Choose Control > Test Movie to test the component in Flash Player.

To export the Dial component:

1. In the Dial.fla file, choose the Dial component in the library and select Export to SWC from the Library options menu.
2. Choose a location to save the SWC file.

   If you save it to the Components folder in the user-level configuration folder, you can reload the Components panel without restarting Flash and the component appears in the panel.

   Note: For information about folder locations, see "Configuration folders installed with Flash" in *Using Flash*.



The completed Dial component

Selecting a parent class

The first thing to decide when creating a component is whether to extend one of the version 2 classes. If you choose to extend a version 2 class, you can either extend a component class (for example, Button, CheckBox, ComboBox, List, and so on) or one of the base classes, UIObject or UIComponent. All the component classes, except the Media components, extend the base classes; if you extend a component class, you automatically inherit from the base classes as well.

The two base classes supply common features for components. By extending these classes, your component begins with a basic set of methods, properties, and events.

You don't have to subclass UIObject or UIComponent or any other classes in the version 2 framework. Even if your component classes inherit directly from the MovieClip class, you can use many powerful component features: export to a SWC file or compiled clip, use built-in live preview, view inspectable properties, and so on. However, if you want your components to work with the Macromedia version 2 components, and use the manager classes, you need to extend UIObject or UIComponent.

The following table briefly describes the version 2 base classes:

Base class	Extends	Description
`mx.core.UIObject`	MovieClip	UIObject is the base class for all graphical objects. It can have shape, draw itself, and be invisible. UIObject provides the following functionality: • Editing styles • Event handling • Resizing by scaling
`mx.core.UIComponent`	UIObject	UIComponent is the base class for all components. UIComponent provides the following functionality: • Creating focus navigation • Creating a tabbing scheme • Enabling and disabling components • Resizing components • Handling low-level mouse and keyboard events.

Understanding the UIObject class

Components based on version 2 of the Macromedia Component Architecture descend from the UIObject class, which is a subclass of the MovieClip class. The MovieClip class is the base class for all classes in Flash that represent visual objects on the screen.

UIObject adds methods that allow you to handle styles and events. It posts events to its listeners just before drawing (the `draw` event is the equivalent of the `MovieClip.onEnterFrame` event), when loading and unloading (`load` and `unload`), when its layout changes (`move`, `resize`), and when it is hidden or revealed (`hide` and `reveal`).

UIObject provides alternate read-only variables for determining the position and size of a component (`width`, `height`, `x`, `y`), and the `move()` and `setSize()` methods to alter the position and size of an object.

The UIObject class implements the following:

- Styles
- Events
- Resize by scaling

Understanding the UIComponent class

The UIComponent class is a subclass of UIObject. It is the base class of all components that handle user interaction (mouse and keyboard input). The UIComponent class allows components to do the following:

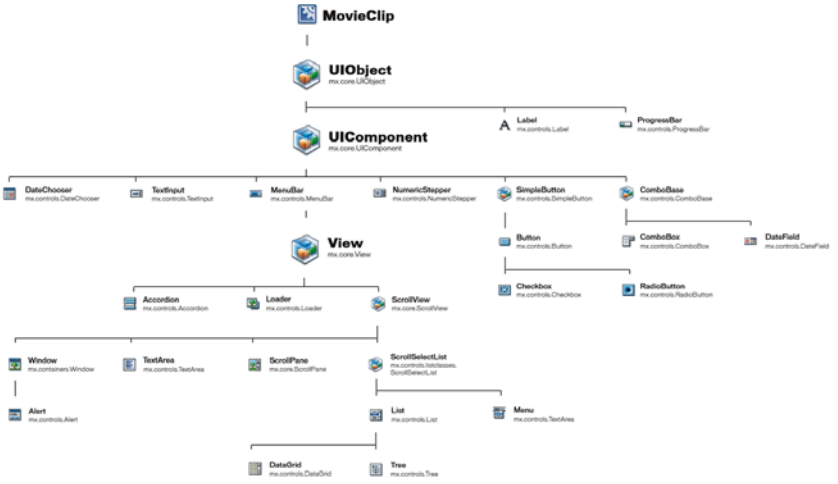
- Receive focus and keyboard input
- Enable and disable components
- Resize by layout

Extending other version 2 classes

To make component construction easier, you can extend any class; you are not required to extend the UIObject or UIComponent class directly. If you extend any other version 2 component's class (except the Media components), you extend UIObject and UIComponent by default. Any component class listed in the Component dictionary can be extended to create a new component class.

For example, if you want to create a component that behaves almost the same as a Button component does, you can extend the Button class instead of re-creating all the functionality of the Button class from the base classes.

The following figure shows the version 2 component hierarchy:



Version 2 component hierarchy

This file is available in the Flash MX 2004\Samples\HelpExamples\images folder.

Extending the MovieClip class

You can choose not to extend a version 2 class and have your component inherit directly from the ActionScript MovieClip class. However, if you want any of the UIObject and UIComponent functionality, you'll have to build it yourself. You can open the UIObject and UIComponent classes (FirstRun/classes/mx/core) to examine how they were built.

Creating a component movie clip

To create a component, you must create a movie clip symbol and link it to the component's class file.

The movie clip has two frames and two layers. The first layer is an Actions layer and has a `stop()` action on Frame 1. The second layer is an Assets layer with two keyframes. Frame 1 contains a bounding box or any graphics that serve as placeholders for the final art. Frame 2 contains all other assets, including graphics and base classes, used by the component.

Inserting a new movie clip symbol

All components are MovieClip objects. To create a new component, you must first insert a new symbol into a new FLA file.

To add a new component symbol:

1. In Flash, create a blank Flash document.
2. Select Insert > New Symbol.

   The Create New Symbol dialog box appears.
3. Enter a symbol name. Name the component by capitalizing the first letter of each word in the component (for example, MyComponent).
4. Select the Movie Clip behavior.
5. Click the Advanced button.

   The advanced settings appear in the dialog box.
6. Select Export for ActionScript.
7. Enter a linkage identifier.
8. In the AS 2.0 Class text box, enter the fully qualified path to the ActionScript 2.0 class.

   The class name should be the same as the component name that appears in the Components panel. For example, the Button component's class is mx.controls.Button.

   Note: Do not include the filename's extension; the AS 2.0 Class text box points to the packaged location of the class and not the file system's name for the file.

   If the ActionScript file is in a package, you must include the package name. This field's value can be relative to the classpath or can be an absolute package path (for example, mypackage.MyComponent).
9. In most cases, you should deselect Export in First Frame (it is selected by default). For more information, see "Component development checklist" on page 956.
10. Click OK.

    Flash adds the symbol to the library and switches to symbol-editing mode. In this mode, the name of the symbol appears above the upper left corner of the Stage, and a cross hair indicates the symbol's registration point.

Editing the movie clip

After you create the new symbol and define the linkages for it, you can define the component's assets in the symbol's Timeline.

A component's symbol should have two layers. This section describes what layers to insert and what to add to those layers.

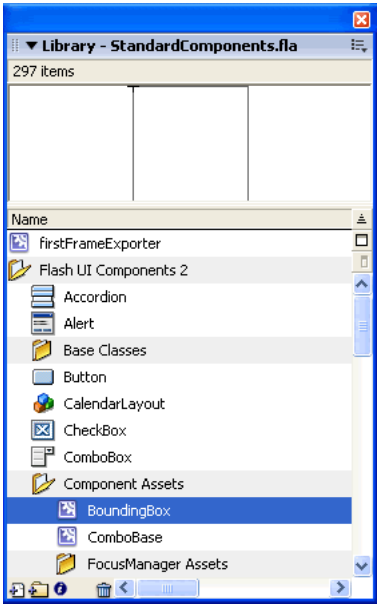
To edit the movie clip:

1. Rename Layer 1 **Actions** and select Frame 1.
2. Open the Actions panel and add a `stop()` function, as follows:

   ```
   stop();
   ```

   Do not add any graphical assets to this frame.
3. Add a Layer named **Assets**.
4. On the Assets layer, select Frame 2 and insert a blank keyframe.

   There are now two blank keyframes in this layer.
5. Do one of the following:
   - If the component has visual assets that define the bounding area, drag the symbols into Frame 1 and arrange them appropriately.
   - If your component creates all its visual assets at runtime, drag a BoundingBox symbol to the Stage in Frame 1, size it correctly, and name the instance **boundingBox_mc**. The symbol is located in the library of the StandardComponents.fla that is located in the First Run/ComponentFLA folder.



6. If you are extending an existing component, place an instance of that component and any other base classes in Frame 2 of the Assets layer.

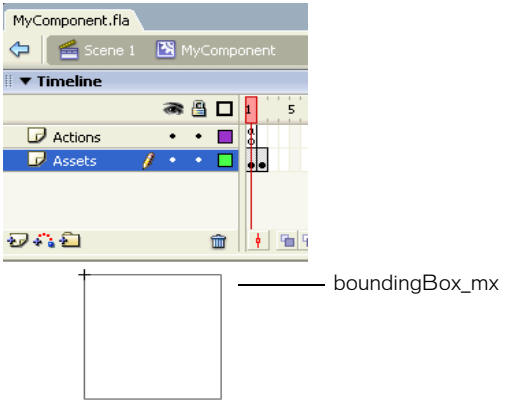
   To do this, select the symbol from the Components panel and drag it onto the Stage. If you are extending a base class, open StandardComponents.fla from the FirstRun/ComponentFLA folder and drag the class from the library to the Stage.

   Note: When you drag UIComponent to the component library, the folder hierarchy in the library is changed. If you plan to use your library again, or use it with other groups of components (such as the version 2 components), you should restructure the folder hierarchy to match the StandardComponents library so that it's organized well and you avoid duplicate symbols.

7. Add any graphical assets used by this component on Frame 2 of your component's Assets layer.

   Any asset that a component uses (whether it's another component or media such as bitmaps) should have an instance placed in the second frame of the Assets layer.

8. Your finished symbol should look something like this:



Defining the movie clip as a component

The movie clip symbol must be linked to an ActionScript class file in the Component Definition dialog box. This is how Flash knows where to look for the component's metatags. (For more information about metatags, see "Adding component metadata" on page 935.) There are other options you can choose in the Component Definition dialog box as well.

To define a movie clip as a component:

1. Select the movie clip in the library, and choose Component Definition from the Library options menu.
2. You must enter an AS 2.0 class.

   If the class is part of a package, enter the full package name.

3. Use other options in the Component Definition dialog box, if desired:

- Click the Plus (+) button to define parameters.

  This is optional. The best practice is to use the metadata Inspectable tag in the component's class file to specify parameters. When an ActionScript 2.0 class is not specified, define the parameters for the component here.

- Specify a custom UI.

  This is a SWF file that plays in the Component inspector. You can embed it in the component FLA file or browse to an external SWF.

- Specify a live preview.

  This is an external or embedded SWF file. You don't have to specify a live preview here; you can add a bounding box to the component movie clip, and Flash creates a live preview for you. See "Creating a component movie clip" on page 927.

- Enter a description.

  The Description field is deprecated in Flash MX 2004 because the Reference panel has been removed. This field is provided for backward compatibility when you save FLA files in the Flash MX format.

- Choose an icon.

  This option specifies a PNG file to use as an icon for the component. If you specify an IconFile metadata tag in the ActionScript 2.0 class file (best practice), this field is ignored.

- Select or deselect Parameters Are Locked in Instances.

  When this option is unselected, users can add parameters to each component instance that differ from the component's parameters. Generally, this setting should be selected. This option provides backward compatibility with Flash MX.

- Specify a tooltip that appears in the Components panel.

Creating the ActionScript class file

All component symbols are linked to an ActionScript 2.0 class file. (For information on linking, see "Creating a component movie clip" on page 927.)

To edit ActionScript class files, you can use Flash, any text editor, or any Integrated Development Environment (IDE).

The external ActionScript class extends another class (whether the class is a version 2 component, a version 2 base class, or the ActionScript MovieClip class). You should extend the class that creates the functionality that is most similar to the component you want to create. You can inherit from (extend) only one class. ActionScript 2.0 does not allow multiple inheritance.

Simple example of a component class file

The following is a simple example of a class file called MyComponent.as. If you were creating this component, you would link this file to the component movie clip in Flash.

This example contains a minimal set of imports, methods, and declarations for a component, MyComponent, that inherits from the UIComponent class. The MyComponents.as file is saved in the myPackage folder.

```
[Event("eventName")]

// Import packages.
import mx.core.UIObject;

// Declare the class and extend from the parent class.
class mypackage.MyComponent extends UIObject {

  // Identify the symbol name that this class is bound to.
  static var symbolName:String = "mypackage.MyComponent";

  // Identify the fully qualified package name of the symbol owner.
  static var symbolOwner:Object = Object(mypackage.MyComponent);

  // Provide the className variable.
  var className:String = "MyComponent";

  // Define an empty constructor.
  function MyComponent() {
  }

  // Call the parent's init method.
  // Hide the bounding box--it's used
  // only during authoring.
  function init():Void {
    super.init();

    boundingBox_mc.width = 0;
    boundingBox_mc.height = 0;
    boundingBox_mc.visible = false;

  }

  function createChildren():Void{
    // Call createClassObject to create subobjects.
    size();
    invalidate();
  }

  function size(){
    // Write code to handle sizing.
    super.size();
    invalidate();
  }

  function draw(){
    // Write code to handle visual representation.
    super.draw();
  }

}
```

Overview of a component class file

The following procedure is an overview of the creation of an ActionScript file for a component. Some steps may be optional, depending on the type of component you create.

To write a component class file:

1. (Optional) Import classes. (See "Importing classes" on page 933).

   This allows you to refer to classes without writing out the package (for example, Button instead of mx.controls.Button).

2. Define the class using the `class` keyword; use the `extend` keyword to extend a parent class. (See "Defining the class and its superclass" on page 933).

3. Define the `symbolName`, `symbolOwner`, and `className` variables. (See "Identifying the class, symbol, and owner names" on page 933).

   These variables are necessary only in version 2 components.

4. Define member variables. (See "Defining variables" on page 934).

   These can be used in getter/setter methods.

5. Define a constructor function. (See "Defining the constructor function" on page 944).

6. Define an `init()` method. (See "Defining the init() method" on page 945).

   This method is called when the class is created if the class extends UIComponent. If the class extends MovieClip, call this method from the constructor function.

7. Define a `createChildren()` method. (See "Defining the createChildren() method" on page 946).

   This method is called when the class is created if the class extends UIComponent. If the class extends MovieClip, call this method from the constructor function.

8. Define a `size()` method. (See "Defining the size() method" on page 947).

   This method is called when the component is resized, if the class extends UIComponent. In addition, this method is called when the component's live preview is resized during authoring.

9. Define a `draw()` method. (See "About invalidation" on page 948).

   This method is called when the component is invalidated, if the class extends UIComponent.

10. Add a Metadata tag and declaration. (See "Adding component metadata" on page 935).

    Adding the tag and declaration causes getter/setter properties to appear in the Property inspector and Component inspector in Flash.

11. Define getter/setter methods. (See "Using getter/setter methods to define parameters" on page 935).

12. (Optional) Create variables for every skin element/linkage used in the component. (See "About assigning skins" on page 950).

    This allows users to set a different skin element by changing a parameter in the component.

Importing classes

You can import class files so that you don't have to write fully qualified class names throughout your code. This can make code more concise and easier to read. To import a class, use the `import` statement at the top of the class file, as in the following:

```
import mx.core.UIObject;
import mx.core.ScrollView;
import mx.core.ext.UIObjectExtensions;

class MyComponent extends UIComponent{
```

You can also use the wildcard character (*) to import all the classes in a given package. For example, the following statement imports all classes in the `mx.core` package:

```
import mx.core.*;
```

If an imported class is not used in a script, the class is not included in the resulting SWF file's bytecode. As a result, importing an entire package with a wildcard does not create an unnecessarily large SWF file.

Defining the class and its superclass

A component class file is defined like any class file. Use the `class` keyword to indicate the class name. The class name must also be the name of the class file. Use the `extends` keyword to indicate the superclass. For more information, see Chapter 10, "Creating Custom Classes with ActionScript 2.0," in *Using ActionScript in Flash*.

```
class MyComponentName extends UIComponent{

}
```

Identifying the class, symbol, and owner names

To help Flash find the proper ActionScript classes and packages and to preserve the component's naming, you must set the `symbolName`, `symbolOwner`, and `className` variables in your component's ActionScript class file.

The `symbolOwner` variable is an Object reference that refers to a symbol. If the component is its own `symbolOwner` or is the `symbolOwner` has been imported, it does not have to be fully qualified.

The following table describes these variables:

Variable	Type	Description
`symbolName`	String	The name of the ActionScript class (for example, ComboBox). This name must match the symbol's Linkage Identifier. This variable must be static.
`symbolOwner`	Object	The fully qualified class name (for example, mypackage.MyComponent). Do not use quotation marks around the `symbolOwner` value, because is is an Object data type. This name must match the AS 2.0 class in the Linkage Properties dialog box. This variable is used in the internal call to the `createClassObject()` method. This variable must be static.
`className`	String	The name of the component class. This does not include the package name and has no corresponding setting in the Flash development environment. You can use the value of this variable when setting style properties.

The following example adds the `symbolName`, `symbolOwner`, and `className` variables to the MyButton class:

```
class MyButton extends mx.controls.Button {
  static var symbolName:String = "MyButton";
  static var symbolOwner = myPackage.MyButton;
  var className:String = "MyButton";
}
```

Defining variables

The following code sample is from Button.as file (mx.controls.Button). It defines a variable, `btnOffset`, to use in the class file. It also defines the variables `__label`, and `__labelPlacement`. The latter two variables are prefixed with a double underscore to prevent name conflicts when they are used in getter/setter methods and ultimately used as properties and parameters in the component. For more information, see "Using getter/setter methods to define parameters" on page 935. See also "Implicit getter/setter methods" in *Using ActionScript in Flash*.

```
/**
* Number used to offset the label and/or icon when button is pressed.
*/
  var  btnOffset:Number = 0;

*@private
* Text that appears in the label if no value is specified.
*/
  var __label:String  = "default value";

/**
*@private
* default label placement
*/
  var __labelPlacement:String  = "right";
```

Using getter/setter methods to define parameters

The simplest way to define a component parameter is to add a public member variable that is Inspectable:

```
[Inspectable(defaultValue="strawberry")]
public var flavorStr:String;
```

However, if code that employs a component modifies the `flavorStr` property, the component typically must perform an action to update itself in response to the property change. For example, if `flavorStr` is set to `"cherry"`, the component might redraw itself with a cherry image instead of the default strawberry image.

For regular member variables, the component is not automatically notified that the member variable's value has changed.

Getter/setter methods are a straightforward way to detect changes to component properties. Instead of declaring a regular variable with `var`, declare getter/setter methods, as follows:

```
private var __flavorStr:String = "strawberry";

[Inspectable(defaultValue="strawberry")]

public function get flavorStr():String{
  return __flavorStr;
}
public function set flavorStr(newFlavor:String) {
  __flavorStr = newFlavor;
  invalidate();
}
```

The `invalidate()` call causes the component to redraw itself with the new flavor. This is the benefit of using getter/setter methods for the `flavorStr` property, instead of a regular member variable. See "Defining the draw() method" on page 947.

To define getter/setter methods, remember these points:

- Precede the method name with `get` or `set`, followed by a space and the property name.
- The variable that stores the property's value cannot have the same name as the getter or setter. By convention, precede the names of the getter and setter variables with two underscores.

Getters and setters are commonly used in conjunction with tags to define properties that are visible in the Property and Component inspectors.

For more information about getters and setters, see "Implicit getter/setter methods" in *Using ActionScript in Flash*.

Adding component metadata

You can add component metadata tags in your external ActionScript class files to tell the compiler about component parameters, data binding properties, and events. Metadata tags are used in the Flash authoring environment for a variety of purposes.

The metadata tags can only be used in external ActionScript class files. You cannot use metadata tags in FLA files.

Metadata is associated with a class declaration or an individual data field. If the value of an attribute is of type String, you must enclose that attribute in quotation marks.

Metadata statements are bound to the next line of the ActionScript file. When defining a component property, add the metadata tag on the line before the property declaration. The only exception is the Event metadata tag. When defining component events, add the metadata tag outside the class definition so that the event is bound to the entire class.

In the following example, the Inspectable tags apply to the `flavorStr`, `colorStr`, and `shapeStr` parameters:

```
[Inspectable(defaultValue="strawberry")]
public var flavorStr:String;
[Inspectable(defaultValue="blue")]
public var colorStr:String;
[Inspectable(defaultValue="circular")]
public var shapeStr:String;
```

In the Property inspector and the Parameters tab of the Component inspector, Flash displays all of these parameters as type String.

Metadata tags

The following table describes the metadata tags you can use in ActionScript class files:

Tag	Description
Inspectable	Exposes a property in the Component inspector and Property inspector. See "About the Inspectable tag" on page 937.
InspectableList	Identifies which subset of inspectable properties should be listed in the Property inspector and Component inspector. If you don't add an InspectableList attribute to your component's class, all inspectable parameters appear in the Property inspector. See "About the InspectableList tag" on page 938.
Event	Defines a component event. See "About the Event tag" on page 939.
Bindable	Reveals a property in the Bindings tab of the Component inspector. See "About the Bindable tag" on page 939.
ChangeEvent	Identifies a event that cause data binding to occur. See "About the ChangeEvent tag" on page 941.
Collection	Identifies a `collection` attribute exposed in the Component inspector. See "About the Collection tag" on page 942.
IconFile	Specifies the filename for the icon that represents this component in the Components panel. See "About the IconFile tag" on page 943.

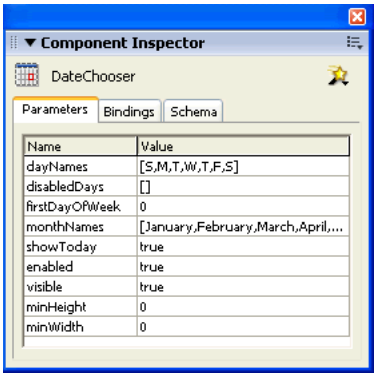
The following sections describe the component metadata tags in more detail.

About the Inspectable tag

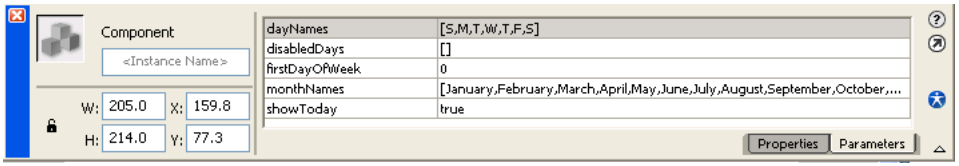
You use the Inspectable tag to specify the user-editable (inspectable) parameters that appear in the Component inspector and Property inspector. This lets you maintain the inspectable properties and the underlying ActionScript code in the same place. To see the component properties, drag an instance of the component onto the Stage and select the Parameters tab of the Component inspector.

Collection parameters are also inspectable. For more information, see "About the Collection tag" on page 942.

The following figure shows the Parameters tab of the Component inspector for the DateChooser component:



Alternatively, you can view a subset of the component properties on the Property inspector Parameters tab.



When determining which parameters to reveal in the authoring environment, Flash uses the Inspectable tag. The syntax for this tag is as follows:

```
[Inspectable(value_type=value[,attribute=value,...])]
property_declaration name:type;
```

The following example defines the `enabled` parameter as inspectable:

```
[Inspectable(defaultValue=true, verbose=1, category="Other")]
var enabled:Boolean;
```

The Inspectable tag also supports loosely typed attributes like this:

```
[Inspectable("danger", 1, true, maybe)]
```

The metadata statement must immediately precede the property's variable declaration in order to be bound to that property.

The following table describes the attributes of the Inspectable tag:

Attribute	Type	Description
`defaultValue`	String or Number	(Optional) A default value for the inspectable property.
`enumeration`	String	(Optional) Specifies a comma-delimited list of legal values for the property.
`listOffset`	Number	(Optional) Added for backward compatibility with Flash MX components. Used as the default index into a List value.
`name`	String	(Optional) A display name for the property. For example, Font Width. If not specified, use the property's name, such as `_fontWidth`.
`type`	String	(Optional) A type specifier. If omitted, use the property's type. The following values are acceptable: • Array • Boolean • Color • Font Name • List • Number • Object • String
`variable`	String	(Optional) Added for backward compatibility with Flash MX components. Specifies the variable that this parameter is bound to.
`verbose`	Number	(Optional) An inspectable property that has the verbose attribute set to 1 does not appear in the Property inspector but does appear in the Component inspector. This is typically used for properties that are not modified frequently.

None of these attributes are required. You can simply have Inspectable as the metadata tag.

All properties of the superclass that are marked Inspectable are automatically inspectable in the current class. Use the InspectableList tag if you want to hide some of these properties for the current class.

About the InspectableList tag

Use the InspectableList tag to specify which subset of inspectable properties should appear in the Property inspector. Use InspectableList in combination with Inspectable so that you can hide inherited attributes for components that are subclasses. If you do not add an InspectableList tag to your component's class, all inspectable parameters, including those of the component's parent classes, appear in the Property inspector.

The InspectableList syntax is as follows:

```
[InspectableList("attribute1"[,...])]
// class definition
```

The InspectableList tag must immediately precede the class definition because it applies to the entire class.

The following example allows the `flavorStr` and `colorStr` properties to be displayed in the Property inspector, but excludes other inspectable properties from the Parent class:

```
[InspectableList("flavorStr","colorStr")]
class BlackDot extends DotParent {
  [Inspectable(defaultValue="strawberry")]
  public var flavorStr:String;
  [Inspectable(defaultValue="blue")]
  public var colorStr:String;
  ...
}
```

About the Event tag

Use the Event tag to define events that the component emits.

This tag has the following syntax:

```
[Event("event_name")]
```

For example, the following code defines a `click` event:

```
[Event("click")]
```

Add the Event statements outside the class definition in the ActionScript file so that the events are bound to the class and not a particular member of the class.

The following example shows the Event metadata for the UIObject class, which handles the `resize`, `move`, and `draw` events:

```
...
import mx.events.UIEvent;
[Event("resize")]
[Event("move")]
[Event("draw")]
class mx.core.UIObject extends MovieClip {
  ...
}
```

To broadcast a particular instance, call the `dispatchEvent()` method. See "Using the dispatchEvent() method" on page 948.

About the Bindable tag

Data binding connects components to each other. You achieve visual data binding through the Bindings tab of the Component inspector. From there, you add, view, and remove bindings for a component.

Although data binding works with any component, its main purpose is to connect user interface components to external data sources, such as web services and XML documents. These data sources are available as components with properties, which you can bind to other component properties.

Use the Bindable tag before a property in an ActionScript class to make the property appear in the Bindings tab in the Component inspector. You can declare a property by using `var` or getter/setter methods. If a property has both a getter and a setter method, you only need to apply the Bindable tag to one.

The Bindable tag has the following syntax:

```
[Bindable "readonly"|"writeonly",type="datatype"]
```

Both attributes are optional.

The following example defines the variable `flavorStr` as a property that is accessible on the Bindings tab of the Component inspector:

```
[Bindable]
public var flavorStr:String = "strawberry";
```

The Bindable tag takes three options that specify the type of access to the property, as well as the data type of that property. The following table describes these options:

Option	Description
`readonly`	Indicates that when you create bindings in the Component inspector, you can only create bindings that use this property as a source. However, if you use ActionScript to create bindings, there is no such restriction. `[Bindable("readonly")]`
`writeonly`	Indicates that when you create bindings in the Component inspector, this property can only be used as the destination of a binding. However, if you use ActionScript to create bindings, there is no such restriction. `[Bindable("writeonly")]`

`type="datatype"`	Indicates the type that data binding uses for the property. The rest of Flash uses the declared type.
	If you do not specify this option, data binding uses the property's data type as declared in the ActionScript code.
	In the following example, data binding will treat `x` as type DataProvider, even though it is really type Object:
	`[Bindable(type="DataProvider")]`
	`var x: Object;`

All properties of all components can participate in data binding. The Bindable tag merely controls which of those properties are available for binding in the Component inspector. If a property is not preceded by the Bindable tag, you can still use it for data binding, but you have to create the bindings using ActionScript.

The Bindable tag is required when you use the ChangeEvent tag. For more information, see the next section.

For information on creating data binding in the Flash authoring environment, see "Data binding (Flash Professional only)" in *Using Flash*.

About the ChangeEvent tag

The ChangeEvent tag tells data binding that the component will generate an event any time the value of a specific property changes. In response to the event, data binding executes any binding that has that property as a source. The component only generates the event if you write appropriate ActionScript code in the component. The event should be included in the list of Event metadata declared by the class.

You can declare a property by using `var` or getter/setter methods. If a property has both a getter and a setter method, you only need to apply the ChangeEvent tag to one.

The ChangeEvent tag has the following syntax:

```
[Bindable]
[ChangeEvent("event")]
property_declaration or getter/setter function
```

In the following example, the component generates the `change` event when the value of the bindable property `flavorStr` changes:

```
[Bindable]
[ChangeEvent("change")]
public var flavorStr:String;
```

When the event that is specified in the metadata occurs, Flash notifies bindings that the property has changed.

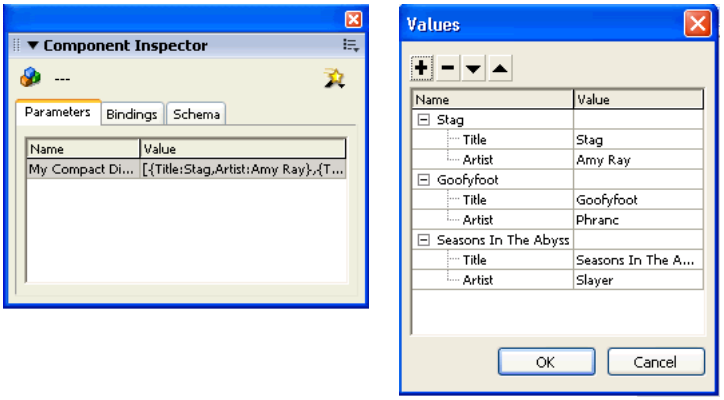
You can register multiple events in the tag, as the following example shows:

```
[ChangeEvent("change1", "change2", "change3")]
```

Any one of those events indicates a change to the property. They do not all have to occur to indicate a change.

About the Collection tag

Use the Collection tag to describe an array of objects that can be modified as a collection of items in the Values dialog box while authoring. The type of the objects is identified by the `collectionItem` attribute. A collection property contains a series of collection items that you define in a separate class. This class is either mx.utils.CollectionImpl or a subclass of it. The individual objects are accessed through the methods of the class identified by the `collectionClass` attribute.



A collection property in the Component inspector and the Values dialog box that appears when you click the magnifying glass.

The syntax for the Collection tag is as follows:

```
[Collection (name="name", variable="varname",
  collectionClass="mx.utils.CollectionImpl",
  collectionItem="coll-item-classname", identifier="string")]
public var varname:mx.utils.Collection;
```

The following table describes the attributes of the Collection tag:

Attribute	Type	Description
`name`	String	(Required) Name that appears in the Component inspector for the collection.
`variable`	String	(Required) ActionScript variable that points to the underlying Collection object (for example, you might name a `Collection` parameter Columns, but the underlying `variable` attribute might be __columns).
`collectionClass`	String	(Required) Specifies the class type to be instantiated for the collection property. This is usually `mx.utils.CollectionImpl`, but it can also be a class that extends `mx.utils.CollectionImpl`.

Attribute	Type	Description
`collectionItem`	String	(Required) Specifies the class of the collection items to be stored within the collection. This class includes its own inspectable properties that are exposed through metadata.
`identifier`	String	(Required) Specifies the name of an inspectable property within the collection item class that Flash MX uses as the default identifier when the user adds a new collection item through the Values dialog box. Each time a user creates a new collection item, Flash MX sets the item name to *identifier* plus a unique index (for example, if `identifier=name`, the Values dialog box displays name0, name1, name2, and so on).

For more information, see "Collection Properties" on page 959.

About the IconFile tag

You can add an icon that represents your component in the Components panel of the Flash authoring environment. For more information, see "Adding an icon" on page 956.

Defining component parameters

When building a component, you can add parameters that define its appearance and behavior. The most commonly used parameters appear as authoring parameters in the Component inspector and Property inspector. You can also set all inspectable and collection parameters with ActionScript. You define these properties in the component class file by using the Inspectable tag (see "About the Inspectable tag" on page 937).

The following example sets several component parameters in the JellyBean class file, and exposes them with the Inspectable tag in the Component inspector:

```
class JellyBean{
  // a string parameter
  [Inspectable(defaultValue="strawberry")]
  public var flavorStr:String;

  // a string list parameter
  [Inspectable(enumeration="sour,sweet,juicy,rotten",defaultValue="sweet")]
  public var flavorType:String;

  // an array parameter
  [Inspectable(name="Flavors", defaultValue="strawberry,grape,orange",
  verbose=1, category="Fruits")]
  var flavorList:Array;

  // an object parameter
  [Inspectable(defaultValue="belly:flop,jelly:drop")]
  public var jellyObject:Object;

  // a color parameter
  [Inspectable(defaultValue="#ffffff")]
  public var jellyColor:Color;

  // a setter
```

```
    [Inspectable(defaultValue="default text")]
    function set text(t:String)

}
```

You can use any of the following types for parameters:

- Array
- Object
- List
- String
- Number
- Boolean
- Font Name
- Color

Note: The JellyBean class is a theoretical example. To see an actual example, look at the Button.as class file that installs with Flash MX 2004 in the First Run/Classes/mx/controls directory.

Defining core functions

You must define five functions in the component class file: the constructor, `init()`, `createChildren()`, `draw()`, and `size()`. When a component extends UIComponent, functions in the class file are called in the following order:

- Constructor
- `init()`
  Once `init()` is called, the `width` and `height` properties are set.
- `createChildren()`
  A frame passes in the Timeline. During this time, the component user can call methods and properties to set up the component.
- `draw()`
- The `size()` method is called whenever a component is resized at runtime.

Defining the constructor function

You can recognize a constructor because it has the same name as the component class. For example, the following code shows the ScrollBar component's constructor function:

```
function ScrollBar() {
}
```

In this case, when a new scroll bar is instantiated, the `ScrollBar()` constructor is called.

Generally, component constructors should be empty. Setting properties in constructors can sometimes lead to overwriting default values, depending on the order of initialization calls.

If your component extends UIComponent or UIObject, Flash automatically calls `init()`, `createChildren()`, and `size()` methods and you can leave your constructor function empty, as shown here:

```
class MyComponent extends UIComponent{
  ...
  // this is the constructor function
  function MyComponent(){
  }
}
```

All version 2 components should define an `init()` function that is called after the constructor has been called. You should place the initialization code in the component's `init()` function. For more information, see the next section.

If your component extends MovieClip, you may want to call an `init()` method, a `createChildren()` method, and a method that lays out your component from the constructor function, as shown here:

```
class MyComponent extends MovieClip{
  ...
  function MyComponent(){
    init()
  }

  function init():Void{
    init();
    createChildren();
    layout();
  }
  ...
}
```

For more information about constructors, see "Constructor functions" in *Using ActionScript in Flash*.

Defining the init() method

Flash calls the `init()` method when the class is created. This method is called once when a component is instantiated and never again.

You should use the `init()` method to do the following:

- Call `super.init()`.

  This is required.

- Make the `boundingBox_mc` invisible.

  ```
  boundingBox_mc.width = 0;
  boundingBox_mc.height = 0;
  boundingBox_mc.visible = false;
  ```

- Create instance member variables.

The `width`, `height`, and `clip` parameters are properly set only after this method is called.

The `init()` method is called from UIObject's constructor, so the flow of control climbs up the chain of constructors until it reaches UIObject. UIObject's constructor calls the `init()` method that is defined on lowest subclass. Each implementation of `init()` should call `super.init()` so that its base class can finish initializing. If you implement an `init()` method and you don't call `super.init()`, the `()init` method is not called on any of the base classes, so they might never be in a usable state.

Defining the createChildren() method

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 `createClassObject()` or `createObject()` to instantiate a subobject of your component.

It's a good idea to call `size()` within the `createChildren()` method to make sure all children are set to the correct size initially. Also, call `invalidate()` within the `createChildren()` method to refresh the screen. (For more information, see "About invalidation" on page 948.)

The `createClassObject()` method has the following syntax:

```
createClassObject(className, instanceName, depth, initObject)
```

The following table describes the parameters:

Parameter	Type	Description
`className`	Object	The name of the class.
`instanceName`	String	The name of the instance.
`depth`	Number	The depth for the instance.
`initObject`	Object	An object that contains initialization properties.

To call `createClassObject()`, you must know what the children are, because you must specify the name and type of the object, plus any initialization parameters in the call to `createClassObject()`.

The following example calls `createClassObject()` to create a new Button object for use inside a component:

```
up_mc.createClassObject(mx.controls.Button, "submit_btn", 1);
```

You set properties in the call to `createClassObject()` by adding them as part of the *`initObject`* parameter. The following example sets the value of the `label` property:

```
form.createClassObject(mx.controls.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);
  size()
  invalidate()
}
```

Defining the draw() method

You can write code in the `draw()` method to create or modify visual elements of a component. In other words, in the `draw()` method, a component draws itself to match its state variables. Since the last `draw()` call, multiple properties or methods may have been called, and you should try to account for all of them in the body of `draw()`.

However, you should not call the `draw()` method directly. Instead, call the `invalidate()` method so that calls to `draw()` can be queued and handled in a batch. This approach increases efficiency and centralizes code. (For more information, see "About invalidation" on page 948.)

Inside the `draw()` method, you can use calls to the Flash drawing API to draw borders, rules, and other graphical elements. You can also set property values and call methods. You can also call the `clear()` method, which removes the visible objects.

In the following example from the Dial component (see "Building your first component" on page 918), the `draw()` method sets the rotation of the needle to the `value` property:

```
function draw():Void {
  super.draw();
  dial.needle._rotation = value;
}
```

Defining the size() method

When a component is resized at runtime using the `componentInstance.setSize()` method, the `size()` function is invoked and passed `width` and `height` properties. You can use the `size()` method in the component's class file to lay out the contents of the component.

At a minimum, the `size()` method should call the superclass's `size()` method (`super.size()`).

In the following example from the Dial component (see "Building your first component" on page 918), the `size()` method uses the `width` and `height` parameters to resize the dial movie clip:

```
function size():Void {
  super.size();
  dial._width  = width;
  dial._height = height;
  invalidate();
}
```

Call the `invalidate()` method inside the `size()` method to tag the component for redraw instead of calling the `draw()` method directly. For more information, see the next section.

About invalidation

Macromedia recommends that a component not update itself immediately in most cases, but that it instead should save a copy of the new property value, set a flag indicating what is changed, and call the `invalidate()` method. (This method indicates that just the visuals for the object have changed, but size and position of subobjects have not changed. This method calls the `draw()` method.)

You must call an invalidation method at least once during the instantiation of your component. The most common place for you to do this is in the `createChildren()` or `layoutChildren()` methods.

Dispatching events

If you want your component to broadcast events other than the events it may inherit from a parent class, you must call the `dispatchEvent()` method in the component's class file.

The `dispatchEvent()` method is defined in the mx.events.EventDispatcher class and is inherited by all components that extend UIObject. (See "EventDispatcher class" on page 415.)

You should also add an Event metadata tag at the top of the class file for each new event. For more information, see "About the Event tag" on page 939.

Note: For information about handling component events in a Flash application, see Chapter 4, "Handling Component Events," on page 55.

Using the dispatchEvent() method

In the body of your component's ActionScript class file, you broadcast events using the `dispatchEvent()` method. The `dispatchEvent()` method has the following syntax:

```
dispatchEvent(eventObj)
```

The *`eventObj`* parameter is an ActionScript object that describes the event (see the example later in this section).

You must declare the `dispatchEvent` function in your code before you call it, as follows:

```
private var dispatchEvent:Function;
```

You must also create an event object to pass to `dispatchEvent()`. The event object contains information about the event that the listener can use to handler the event.

You can explicitly build an event object before dispatching the event, as the following example shows:

```
var eventObj = new Object();
eventObj.type = "myEvent";
eventObj.target = this;
dispatchEvent(eventObj);
```

You can also use a shortcut syntax that sets the value of the `type` property and the `target` property and dispatches the event in a single line:

```
ancestorSlide.dispatchEvent({type:"revealChild", target:this});
```

In the preceding example, setting the `target` property is optional, because it is implicit.

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 takes a `detail` property in addition to the `type` and `target` properties. For more information, see the event descriptions in Chapter 6, "Components Dictionary," on page 91.

Common events

The following table lists the common events that are broadcast by various classes. Every component should broadcast these events if they make sense for that component. This is not a complete list of events for all components, just ones that are likely to be reused by other components. Even though some events specify no parameters, all events have an implicit parameter: a reference to the object broadcasting the event.

Event	Use
`click`	Used by the Button component, or whenever a mouse click has no other meaning.
`change`	Used by List, ComboBox, and other text-entry components.
`scroll`	Used by ScrollBar and other controls that cause scrolling (scroll "bumpers" on a scrolling pop-up menu).

In addition, because of inheritance from the base classes, all components broadcast the following events:

UIComponent event	Description
`load`	The component is creating or loading its subobjects.
`unload`	The component is unloading its subobjects.
`focusIn`	The component now has the input focus. Some HTML-equivalent components (ListBox, ComboBox, Button, Text) might also broadcast `focus`, but all broadcast `DOMFocusIn`.
`focusOut`	The component has lost the input focus.
`move`	The component has been moved to a new location.
`resize`	The component has been resized.

The following table describes common key events:

Key events	Description
`keyDown`	A key is pressed. The `code` property contains the key code and the `ascii` property contains the ASCII code of the key pressed. Do not check with the low-level Key object, because the event might not have been generated by the Key object.
`keyUp`	A key is released.

About assigning skins

A user interface (UI) component is composed entirely of attached movie clips. This means that all assets for a UI component can be external to the UI component movie clip, so they can be used by other components. For example, if your component needs check box functionality, you can reuse the existing CheckBox component assets.

The CheckBox component uses a separate movie clip to represent each of its states (FalseUp, FalseDown, Disabled, Selected, and so on). However, you can associate custom movie clips—called *skins*—with these states. At runtime, the old and new movie clips are exported in the SWF file. The old states simply become invisible to give way to the new movie clips. The ability to change skins during authoring and at runtime is called *skinning*.

To skin components, create a variable for every skin element (movie clip symbol) used in the component and set it to the symbol's linkage ID. This lets a developer set a different skin element just by changing a parameter in the component, as shown here:

```
var falseUpIcon = "mySkin";
```

The following example shows the skin variables for the various states of the CheckBox component:

```
var falseUpSkin:String  = "";
var falseDownSkin:String  = "";
var falseOverSkin:String = ""
var falseDisabledSkin:String = "";
var trueUpSkin:String = "";
var trueDownSkin:String = "";
var trueOverSkin:String = "";
var trueDisabledSkin:String = "";
var falseUpIcon:String = "CheckFalseUp";
var falseDownIcon:String = "CheckFalseDown";
var falseOverIcon:String = "CheckFalseOver";
var falseDisabledIcon:String = "CheckFalseDisabled";
var trueUpIcon:String = "CheckTrueUp";
var trueDownIcon:String = "CheckTrueDown";
var trueOverIcon:String = "CheckTrueOver";
var trueDisabledIcon:String = "CheckTrueDisabled";
```

About styles

You can use styles to register all the graphics in your component with a class and let that class control the color scheme of the graphics at runtime. No special code is necessary in the component implementations to support styles. Styles are implemented entirely in the base classes (UIObject and UIComponent) and skins.

To add a new style to a component, call `getStyle("styleName")` in the component class. If the style has been set on an instance, on a custom style sheet, or on the global style sheet, the value is retrieved. If not, you may need to install a default value for the style on the global style sheet.

For more information about styles, see "Using styles to customize component color and text" on page 67.

Registering skins to styles

The following example creates a component called Shape. This component displays a shape that is one of two skins: a circle or a square. The skins are registered to the `themeColor` style.

To register a skin to a style:

1. Create a new ActionScript file and copy the following code into it:

```
import mx.core.UIComponent;

class Shape extends UIComponent{

  static var symbolName:String = "Shape";
  static var symbolOwner:Object = Shape;
  var className:String = "Shape";

  var themeShape:String = "circle_skin"

  function Shape(){
  }

  function init(Void):Void{
    super.init();
  }

  function createChildren():Void{
    setSkin(1, themeShape);
    super.createChildren();
  }
}
```

2. Save the file as **Shape.as**.
3. Create a new Flash document and save it as **Shape.fla** in the same folder as Shape.as
4. Draw a circle on the Stage, select it, and press F8 to convert it to a movie clip.

   Give the circle the name and linkage identifier **circle_skin**.
5. Open the circle_skin movie clip and place the following ActionScript on Frame 1 to register the symbol with the style name `themeColor`:

```
mx.skins.ColoredSkinElement.setColorStyle(this, "themeColor");
```

6. Create a new movie clip for the component.

   Give the movie clip the name and linkage identifier **Shape**.
7. Create two layers. Place a `stop()` action in the first frame of the first layer. Place the symbol circle_skin in the second frame.

   This is the component movie clip. For more information, see "Creating a component movie clip" on page 927.
8. Open StandardComponents.fla as an external library, and drag the UIComponent movie clip to the Stage on the second frame of the Shape movie clip (with circle_skin).
9. Close StandardComponents.fla.

10. Select the Shape movie clip in the library, select Component Definition from the Library options menu, and enter the AS 2.0 class **Shape**.
11. Test the movie clip with the Shape component on the Stage.

    To change the theme color, set the style on the instance. The following code changes the color of a Shape component with the instance name `shape` to red:

    ```
    shape.setStyle("themeColor",0xff0000);
    ```

12. Draw a square on the Stage and convert it to a movie clip.

    Give it the linkage name **square_skin**, and make sure Export in First Frame is selected.

    Note: Because the movie clip isn't placed in the component, Export in First Frame must be selected so that the skin is available before initialization.

13. Open the square_skin movie clip and place the following ActionScript on Frame 1 to register the symbol with the style name `themeColor`:

    ```
    mx.skins.ColoredSkinElement.setColorStyle(this, "themeColor");
    ```

14. Place the following code on the instance of the Shape component on the Stage in the main Timeline:

    ```
    onClipEvent(initialize){
      themeShape = "square_skin";
    }
    ```

15. Test the movie clip with Shape on the Stage. The result should display a red square.

Registering a new style name

If you have created a new style name and it is a color style, add the new name to the `colorStyles` object in the StyleManager.as file (First Run\Classes\mx\styles\StyleManager.as). This example adds the `shapeColor` style:

```
// initialize set of inheriting color styles
  static var colorStyles:Object =
  {
    barColor: true,
    trackColor: true,
    borderColor: true,
    buttonColor: true,
    color: true,
    dateHeaderColor: true,
    dateRollOverColor: true,
    disabledColor: true,
    fillColor: true,
    highlightColor: true,
    scrollTrackColor: true,
    selectedDateColor: true,
    shadowColor: true,
    strokeColor: true,
    symbolBackgroundColor: true,
    symbolBackgroundDisabledColor: true,
    symbolBackgroundPressedColor: true,
    symbolColor: true,
```

```
    symbolDisabledColor: true,
    themeColor:true,
    todayIndicatorColor: true,
    shadowCapColor:true,
    borderCapColor:true,
    focusColor:true,
    shapeColor:true
  };
```

Register the new style name to the circle and square skins on Frame 1 of each skin movie clip, as follows:

```
mx.skins.ColoredSkinElement.setColorStyle(this, "shapeColor");
```

The color can be changed with the new style name by setting the style on the instance, as shown here:

```
shape.setStyle("shapeColor",0x00ff00);
```

Exporting and distributing a component

Flash MX 2004 and Flash MX Professional 2004 export components as component packages (SWC files). Components may be distributed as SWC files or as FLA files. (See the article on Macromedia DevNet at www.macromedia.com/support/flash/applications/creating_comps/creating_comps12.html for information about distributing a component as a FLA.)

The best way to distribute a component is to export is as a SWC file, because SWC files contain all the ActionScript, SWF files, and other optional files needed to use the component. SWC files are also useful if you are working at the same time on a component and the application that uses the component.

SWC files can be used to distribute components for use in Macromedia Flash MX 2004, Macromedia Dreamweaver MX 2004, and Macromedia Director MX 2004.

Whether you're developing a component for someone else's use, or for your own, it's important to test the SWC file as an ongoing part of component development. For example, problems can arise in a component's SWC file that don't appear in the FLA file.

This section describes a SWC file and explains how to import and export SWC files in Flash.

Understanding SWC files

A SWC file is a zip-like file (packaged and expanded by means of the PKZIP archive format) generated by the Flash authoring tool.

The following table describes the contents of a SWC file:

File	Description
catalog.xml	(Required) Lists the contents of the component package and its individual components, and serves as a directory to the other files in the SWC file.
ActionScript (AS) files	If you create the component with Flash MX Professional 2004, the source code is one or more ActionScript files that contain a class declaration for the component. The compiler uses the source code for type checking when a component is extended. The AS file is not compiled by the authoring tool because the compiled bytecode is already in the implementing SWF file. The source code may contain intrinsic class definitions that contain no function bodies and are provided purely for type checking.
SWF files	(Required) SWF files that implement the components. One or more components can be defined in a single SWF file. If the component is created with Flash MX 2004, only one component is exported per SWF file.
Live Preview SWF files	(Optional) If specified, these SWF files are used for live preview in the authoring tool. If omitted, the SWF files that implement the component are used for live preview instead. The Live Preview SWF file can be omitted in nearly all cases; it should be included only if the component's appearance depends on dynamic data (for example, a text field that shows the result of a web service call).
SWD file	(Optional) A SWD file corresponding to the implementing SWF file that allows you to debug the SWF file. The filename is always the same as that of the SWF file, but with the extension .swd.
PNG file	(Optional) A PNG file containing the 18 x 18, 8-bit-per-pixel icon that you use to display a component icon in the authoring tool user interfaces. If no icon is supplied, a default icon is displayed. (See "Adding an icon" on page 956.)
Property inspector SWF file	(Optional) A SWF file that you use as a custom Property inspector in the authoring tool. If you omit this file, the default Property inspector is displayed to the user.

You can optionally include other files in the SWC file, after you generate it from the Flash environment. For example, you might want to include a Read Me file, or the FLA file if you want users to have access to the component's source code. To add additional files, use the Macromedia Extension Manager (see www.macromedia.com/exchange/em_download/).

SWC files are expanded into a single directory, so each component must have a unique file name to prevent conflicts.

Exporting SWC files

Flash MX 2004 and Flash MX Professional 2004 provide the ability to export SWC files by exporting a movie clip as a SWC file. When exporting a SWC file, Flash reports compile-time errors as if you were testing a Flash application.

There are two reasons to export a SWC file:

- To distribute a finished component
- To test during development

Exporting a SWC for a completed component

You can export components as SWC files that contain all the ActionScript, SWF files, and other optional files needed to use the component.

To export a SWC file for a completed component:

1. Select the component movie clip in the Flash library.
2. Select Export SWC File from the Library options menu.
3. Save the SWC file.

Testing a SWC during development

At different stages of development, it's a good idea to export the component as a SWC and test it in an application. If you export the SWC to the Components folder in your user-level Configuration folder, you can reload the Components panel without quitting and restarting Flash.

To test a SWC during development:

1. Select the component movie clip in the Flash library.
2. Select Export SWC File from the Library options menu.
3. Browse to the Components folder in your user-level configuration folder.

   Configuration/Components

   Note: For information about the location of the folder, see "Configuration folders installed with Flash" in *Using Flash*.
4. Save the SWC file.
5. Choose Reload from the Components panel's options menu.

   The component appears in the Component panel.
6. Drag the component from the Component panel into a document.

Importing component SWC files into Flash

When you distribute your components to other developers, you can include the following instructions so that they can install and use them immediately.

To import a SWC file:

1. Copy the SWC file into the First Run/Components directory.
2. Restart Flash.

   The component's icon should appear in the Components panel.

Adding the finishing touches

After you create the component and prepare it for packaging, you can add an icon and a tool tip. You can also refer to the "Component Development Checklist" to make sure you completed all the necessary steps.

Adding an icon

You can add an icon that represents your component in the Components panel of the Flash authoring environment.

To add an icon for your component:

1. Create a new image.

   The image must measure 18 pixels square, and you must save it in PNG format. It must be 8-bit with alpha transparency, and the upper left pixel must be transparent to support masking.

2. Add the following definition to your component's ActionScript class file before the class definition:

   ```
   [IconFile("component_name.png")]
   ```

3. Add the image to the same directory as the FLA file. When you export the SWC file, Flash includes the image at the root level of the archive.

Adding a tooltip

Tooltips appear when a user rolls the mouse over your component name or icon in the Components panel of the Flash authoring environment.

You can define a tooltip in the Component Definition dialog box. You can access this dialog box from the Library options menu of the component's FLA file.

To add a tooltip in the Component Definition dialog box:

1. Open the Component Definition dialog box.
2. Select the Display in Components Panel option.
3. Enter the text of your tooltip in the Tool Tip Text text box.

Component development checklist

When you design a component, use the following practices:

- Keep the file size as small as possible.
- Make your component as reusable as possible by generalizing functionality.
- Use the RectBorder class (mx.skins.halo.RectBorder) rather than graphical elements to draw borders around objects. (See "RectBorder class" on page 647.)
- Use tag-based skinning.
- Define the `symbolName`, `symbolOwner`, and `className` variables.
- Assume an initial state. Because style properties are now on the object, you can set initial settings for styles and properties so your initialization code does not have to set them when the object is constructed, unless the user overrides the default state.

- When defining the symbol, do not select the Export in First Frame option unless absolutely necessary. Flash loads the component just before it is used in your Flash application, so if you select this option, Flash preloads the component in the first frame of its parent. The reason you typically do not preload the component in the first frame is for considerations on the web: the component loads before your preloader begins, defeating the purpose of the preloader.
- Avoid multiple frame movie clips (except for the two-frame Assets layer).
- Always implement `init()` and `size()` methods and call `Super.init()` and `Super.size()` respectively, but otherwise keep them lightweight.
- Avoid absolute references, such as `_root.myVariable`.
- Use `createClassObject()` instead of `attachMovie()`.
- Use `invalidate()` and `invalidateStyle()` to invoke the `draw()` method instead of calling `draw()` explicitly.

APPENDIX
Collection Properties

This appendix explains how to create a collection property in a component.

The Collection class is a helper class used to manage a group of zero or more related objects, called collection items. If you define a property of your component as a collection and make it available to users through the Component inspector, they can add, delete, and modify collection items in the Values dialog box while authoring.

You define collections and collection items as follows:

- Define a collection property using the Collection metadata tag in a component's ActionScript file. For more information, see "About the Collection tag" on page 942.
- Define a collection item as a class in a separate ActionScript file that contains its own inspectable properties.

Collections make it easier for you to manage groups of related items programmatically. (In previous versions of Flash, component authors managed groups of related items through multiple programmatically synchronized arrays).

In addition to the Values dialog box, Flash provides the Collection and Iterator interfaces to manage Collection instances and values programmatically. See "Collection interface (Flash Professional only)" on page 169 and "Iterator interface (Flash Professional only)" on page 441.

This appendix contains the following sections:

Defining a collection property

You define a collection property by using the Collection tag in a component's ActionScript file. For more information, see "About the Collection tag" on page 942.

Note: This section assumes that you know how to create components and inspectable component properties.

To define a collection property:

1. Create a FLA file for your component.

   See "Creating a component movie clip" on page 927.
2. Create an ActionScript class.

   See "Creating the ActionScript class file" on page 930.
3. In the ActionScript class, insert a Collection metadata tag.

   For more information, see "About the Collection tag" on page 942.
4. Define `get` and `set` methods for the collection in the component's ActionScript file.
5. Add the utilities classes to your FLA file by selecting Windows > Other Panels > Common Libraries > Classes and dragging UtilsClasses into the component's library.

   UtilsClasses contains the mx.utils.* package for the Collection interface.

   Note: Because UtilsClasses is associated with the FLA file, not the ActionScript class, Flash throws compiler errors when you check syntax while viewing the component's ActionScript class.
6. Code a class that contains the collection item properties.

   See "Defining the class for a collection item" on page 962.

Simple collection example

The following is a simple example of a component class file called MyShelf.as. This example contains a collection property along with a minimal set of imports, methods, and declarations for a component that inherits from the UIObject class.

If you import mx.utils.* in this example, the class names from mx.utils no longer need to be fully qualified. For instance, mx.utils.Collection can be written Collection.

```
import mx.utils.*;
// standard class declaration
```

```
class MyShelf extends mx.core.UIObject
{
// required variables for all classes
  static var symbolName:String = "MyShelf";
  static var symbolOwner:Object = Object(MyShelf);
  var className:String = "MyShelf";

// the Collection metadata tag and attributes
    [Collection(variable="myCompactDiscs",name="My Compact
  Discs",collectionClass="mx.utils.CollectionImpl",
  collectionItem="CompactDisc", identifier="Title")]

// get and set methods for the collection
  public function get MyCompactDiscs():mx.utils.Collection
  {
    return myCompactDiscs;
  }
  public function set MyCompactDiscs(myCDs:mx.utils.Collection):Void
  {
    myCompactDiscs = myCDs;
  }

  // private class member
  private var myCompactDiscs:mx.utils.Collection;


// You must code a reference to the collection item class
// to force the compiler to include it as a dependency
// within the SWC
  private var collItem:CompactDisc;

// You must code a reference to the mx.utils.CollectionImpl class
// to force the compiler to include it as a dependency
// within the SWC
  private var coll:mx.utils.CollectionImpl;

  // required methods for all classes
  function init(Void):Void {
    super.init();
  }
  function size(Void):Void {
    super.size();
  }
}
```

To create a FLA file to accompany this class for testing purposes:

1. In Flash, select File > New and create a Flash document.
2. Select Insert > New Symbol. Give it the name, linkage identifier, and AS 2.0 class name **MyShelf**.
3. Deselect Export in First Frame and click OK.
4. Select the MyShelf symbol in the library and choose Component Definition from the Library options menu. Enter the ActionScript 2.0 class name **MyShelf**.

5. Choose Window > Other Panels > Common Libraries > Classes, and drag UtilClasses to the library of MyShelf.fla.
6. In the MyShelf symbol's Timeline, name one layer **Assets**. Create another layer and name it **Actions**.
7. Place a `stop()` function on Frame 1 of the Actions layer.
8. Select Frame 2 of the Assets layer and select Insert > Timeline > Keyframe.
9. Open StandardComponents.fla from the Configuration/ComponentFLA folder, and drag an instance of UIObject to the Stage of MyShelf in Frame 2 of the Assets layer.

   You must include UIObject in the component's FLA file because, as you can see in the above class file, MyShelf extends UIObject.
10. In Frame 1 of the Assets layer, draw a shelf.

    This can be a simple rectangle; it's just a visual representation of the MyShelf component to use for learning purposes.
11. Select the MyShelf movie clip in the library, and select Convert to Compiled Clip.

    This allows you to drag the MyShelf SWF file (the compiled clip that's added to the library) into the MyShelf.fla file to test the component. Whenever you recompile the component, a Resolve Library Conflict dialog box appears, because an older version of the component already exists in the library. Choose to replace existing items.

    Note: You should have already created the CompactDisc class; otherwise, you'll get compiler errors when converting to a compiled clip.

Defining the class for a collection item

You code the properties for a collection item in a separate ActionScript class, which you define as follows:

- Define the class such that it does not extend UIObject or UIComponent.
- Define all properties using the Inspectable tag.
- Define all properties as variables. Do not write `get` and `set` (getter/setter) methods.

The following is a simple example of a collection item class file called CompactDisc.as.

```
class CompactDisc{
   [Inspectable(type="String", defaultValue="Title")]
   var title:String;
   [Inspectable(type="String", defaultValue="Artist")]
   var artist:String;
}
```

To view the CompactDisc.as class file, see "Simple collection example" on page 960.

Accessing collection information programmatically

Flash provides programmatic access to collection data through the Collection and Iterator interfaces. The Collection interface lets you add, modify, and remove items in a collection. The Iterator interface allows you to loop through the items in a collection.

There are two scenarios in which to use the Collection and Iterator interfaces:

- "Accessing collection information in a component class (AS) file" on page 963
- "Accessing collection items at runtime in a Flash application" on page 964

Advanced developers can also create, populate, access, and delete collections programmatically; for more information, see "Collection interface (Flash Professional only)" on page 169.

Accessing collection information in a component class (AS) file

In a component's class file, you can write code that interacts with collection items defined during authoring or at runtime.

To access collection item information in a component class file, use any of the following approaches:

- The Collection tag includes a `variable` attribute, for which you specify a variable of type mx.utils.Collection. Use this variable to access the collection, as shown in this example:

```
[Collection(name="LinkButtons", variable="__linkButtons",
  collectionClass="mx.utils.CollectionImpl", collectionItem="ButtonC",
  identifier="ButtonLabel")]
public var __linkButtons:mx.utils.Collection;
```

- Access an iterator for the collection items by calling the `Collection.getIterator()` method, as shown in this example:

```
var itr:mx.utils.Iterator = __linkButtons.getIterator();
```

- Use the Iterator interface to step through the items in the collection. The `Iterator.next()` method returns type Object, so you must cast the result to the type of your collection item, as shown in this example:

```
while (itr.hasNext()) {
  var button:ButtonC = ButtonC(itr.next());
  ...
}
```

- Access collection item properties, as appropriate for your application, as shown in this example:

```
item.label =  button.ButtonLabel;

if (button.ButtonLink != undefined) {
  item.data = button.ButtonLink;
}
else {
  item.enabled = false;
}
```

Accessing collection items at runtime in a Flash application

If a Flash application uses a component that has a collection property, you can access the collection property at runtime. This example adds several items to a collection property using the Values dialog and displays them at runtime using the Collection and Iterator APIs.

To access collection items at runtime:

1. Open the MyShelf.fla file that you created earlier.

   See "Simple collection example" on page 960.

   This example builds on the MyShelf component and CompactDisc collection.

2. Open the Library panel, drag the component onto the Stage, and give it an instance name.

   This example uses the instance name myShelf.

3. Select the component, open the Component inspector, and display the Parameters tab. Click the line that contains the collection property, and click the magnifying glass to the right of the line. Flash displays the Values dialog box.

4. Use the Values dialog box to enter values into the collection property.

5. With the component selected on the Stage, open the Actions panel and enter the following code (which must be attached to the component):

```
onClipEvent (mouseDown) {
  import mx.utils.Collection;
  import mx.utils.Iterator;
  var myColl:mx.utils.Collection;
  myColl = _parent.myShelf.MyCompactDiscs;

  var itr:mx.utils.Iterator = myColl.getIterator();
  while (itr.hasNext()) {
    var cd:CompactDisc = CompactDisc(itr.next());
    var title:String = cd.Title;
    var artist:String = cd.Artist;
    trace("Title: " + title + " Artist: " + artist);
  }
}
```

   To access a collection, use the syntax *componentName*.*collectionVariable*; to access an iterator and step through the collection items, use *componentName*.*collectionVariable*.getIterator().

6. Select Control > Test Movie and click the shelf to see the collection data in the Output panel.

Exporting components that have collections to SWC files

When you distribute a component that has a collection, the SWC file must contain the following dependent files:

- Collection interface
- Collection implementation class
- Collection item class
- Iterator interface

Of these files, your code typically uses the Collection and Iterator interfaces, which marks them as dependent classes. Flash automatically includes dependent files in the SWC file and output SWF file.

However, the collection implementation class (mx.utils.CollectionImpl) and component-specific collection item class are not automatically included in the SWC file.

To include the collection implementation class and collection item class in the SWC file, you define private variables in your component's ActionScript file, as the following example shows:

```
// collection item class
private var collItem:CompactDisc;
// collection implementation class
private var coll:mx.utils.CollectionImpl;
```

For more information on SWC files, see "Understanding SWC files" on page 953.

Using a component that has a collection property

When you use a component that includes a collection property, you typically use the Values dialog box to establish the items in the collection.

To use a component that includes a collection property:

1. Add the component to the Stage.
2. Use the Property inspector to name the component instance.
3. Open the Component inspector and display the Parameters tab.
4. Click the line that contains the collection property, and click the magnifying glass to the right of the line.

   Flash displays the Values dialog box.
5. Click the Add (+) button and define a collection item.
6. Click the Add (+), Delete (-), and arrow buttons to add, modify, move, and delete collection items.
7. Click OK.

For information on accessing a collection programmatically, see "Accessing collection items at runtime in a Flash application" on page 964.

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X