MX

macromedia®
FLASH™MX
2004

Using Components

Acknowledgments

Director: Erick Vera

Project Management: Stephanie Gowin, Barbara Nelson

Writing: Jody Bleyle, Mary Burger, Kim Diezel, Stephanie Gowin, Dan Harris, Barbara Herbert, Barbara Nelson, Shirley Ong, Tim Statler

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, Jeff Harmon

First Edition: October 2003

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 while authoring in Macromedia Flash, and ActionScript APIs 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 (v2) 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 v2 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 v2 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.

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 Macromedia Flash, writing ActionScript, and Macromedia Flash Player.

This book assumes that you already have Flash MX 2004 or Flash MX Professional 2004 installed and know how to use it. Before using components, you should complete the lesson "Create a user interface with components" (select Help > How Do I > Quick Tasks > Create a user interface with components).

If you want to write as little ActionScript as possible, you can drag components into a document, set their parameters in the Property inspector or in the Components Inspector panel, and attach an `on()` handler directly to a component in the Actions panel to handle component events.

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

For more information, see Chapter 2, "Working with Components," on page 15.

System requirements

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

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. To install components downloaded from the Exchange, download and install the Macromedia Extension Manager.

Any component, whether it's a SWC file or a FLA file (see "About compiled clips and SWC files" on page 14), 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>\First Run\Components (Windows)
- HD/Applications/Macromedia Flash MX 2004/First Run/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.

About the documentation

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

- For information about Macromedia Flash, see *Getting Started with Flash* (or Getting Started Help), Using Flash Help, ActionScript Reference Guide Help, and ActionScript Dictionary Help.
- For information about accessing web services with Flash applications, see *Using Flash Remoting*.

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 an ActionScript parameter.
- **Bold font** indicates a verbatim entry.

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 book:

at runtime When the code is running in Flash Player.

while authoring While 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, 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 ActionScript Reference Guide Help and ActionScript Dictionary Help.

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 provide any functionality that its creator can imagine. 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 (v2) of the Macromedia Component Architecture, which allows you to easily and quickly build robust applications with a consistent appearance and behavior. The v2 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 8. You can also download components built by members of the Flash community at the Macromedia Exchange.

Benefits of v2 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 panel.

Members of the Flash community can use the Macromedia 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 v2 component architecture share core functionality such as styles, event handling, skinning, focus management, and depth management. When you add the first v2 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 v1 components to v2 components, see "Upgrading version 1 components to version 2 architecture" on page 25.

Categories of components

Components included with Flash MX 2004 and Flash MX Professional 2004 fall into five categories: user interface components, data components, media components, managers, and screens. User interface components allow you to interact with an application; for example, the RadioButton, CheckBox, and TextInput components are user interface controls. Data components allow you to load and manipulate information from data sources; the WebServiceConnector and XMLConnector components are data components. Media components allow you to play back and control streaming media; MediaController, MediaPlayback, and MediaDisplay are the media components. Managers are nonvisual components that allow you to manage a feature, such as focus or depth, in an application; the FocusManager, DepthManager, PopUpManager, and StyleManager are the manager components included with Flash MX 2004 and Flash MX Professional 2004. 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 4, "Components Dictionary," on page 43.

Component architecture

You can use the Property inspector or the Component Inspector panel 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 using version 2 (v2) of the Macromedia Component Architecture. Version 2 components are supported by Flash Player 6 and Flash Player 7. These components are not always compatible with components built using version 1 (v1) architecture (all components released before Flash MX 2004). Also, v1 components are not supported by Flash Player 7. For more information, see "Upgrading version 1 components to version 2 architecture" on page 25.

V2 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 have built-in live previews and cannot be edited, but you can change their parameters in the Property inspector and Component Inspector panel, just as you would with any component. For more information, see "About compiled clips and SWC files" on page 14.

V2 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 ActionScript Reference Guide Help.

All 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 4, "Components Dictionary," on page 43.

All components also use the same event model, CSS-based styles, and built-in skinning mechanism. For more information on styles and skinning, see Chapter 3, "Customizing Components," on page 27. For more information on event handling, see Chapter 2, "Working with Components," on page 15.

What's new in v2 components

Component Inspector panel allows you to change component parameters while authoring in both Macromedia Flash and Macromedia Dreamweaver. (See "Components in the Component Inspector panel and Property inspector" on page 16.)

Listener event model allows listener objects of functions to handle events. (See "About component events" on page 22.)

Skin properties allow you to load states only when needed. (See "About skinning components" on page 36.)

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 27.)

Themes allow you to drag a new look onto a set of components. (See "About themes" on page 34.)

Halo theme provides a ready-made, responsive, and flexible user interface for applications.

Manager classes provide an easy way to handle focus and depth in a application. (See "Creating custom focus navigation" on page 24 and "Managing component depth in a document" on page 25.)

Base classes UIObject and UIComponent provide core functionality to all components. (See "UIComponent" on page 553 and "UIObject" on page 562.)

Packaging as a SWC file allows easy distribution and concealable code. See Chapter 5, "Creating Components," on page 639.

Built-in data binding is available through the Component Inspector panel. For more information about this feature, press the Help Update button.

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 5, "Creating Components," on page 639 and ActionScript Reference Guide Help.

About compiled clips and SWC files

A compiled clip is used to pre-compile complex symbols in a Flash document. For example, a movie clip with a lot of ActionScript code that doesn't change often could be turned into a compiled clip. As a result, both Test Movie and Publish would require less time to execute.

A SWC file is the file type for saving and distributing components. 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.

For more information about SWC files, see Chapter 5, "Creating Components," on page 639.

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 4, "Components Dictionary," on page 43. 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. For more information, see "Creating Accessible Content" in Using Flash Help.

Most components built by Macromedia are also navigable by the keyboard. Each component's entry in Chapter 4, "Components Dictionary," on page 43 indicates whether or not you can control the component with the keyboard.

CHAPTER 2
Working with Components

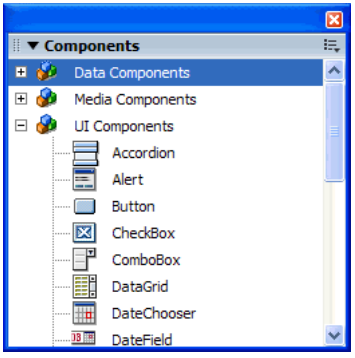
There are various ways to work with components in Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004. You use the Components panel to view components and add them to a document during authoring. Once a component has been added to a document, you can view its properties in the Property inspector or in the Component Inspector panel. Components can communicate with other components by listening to their events and handling them with ActionScript. You can also manage the component depth in a document and control when a component receives focus.

The Components panel

All components are stored in the Components panel. When you install Flash MX 2004 or Flash MX Professional 2004 and launch it for the first time, the components in the Macromedia\Flash 2004\en\First Run\Components (Windows) or Macromedia Flash 2004/en/First Run/Components (Macintosh) folder are displayed in the Components panel.

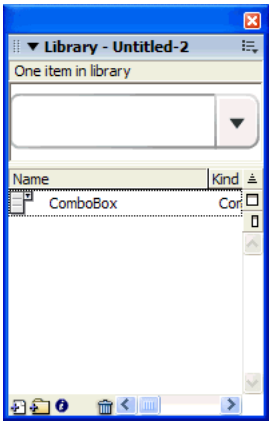
To display the Components panel:

- Select Window > Development Panels > Components.



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 "Working with SWC files and compiled clips" on page 18.

Components in the Component Inspector panel and Property inspector

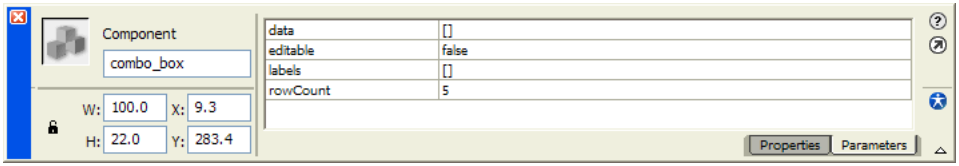
After you add an instance of a component to a Flash document, you use the Property inspector to set and view information for the instance. You create an instance of a component by dragging it from the Components panel onto the Stage; then you name the instance in the Property inspector and specify the parameters for the instance using the fields on the Parameters tab. You can also set parameters for a component instance using the Component Inspector panel. It doesn't matter which panel you use to set parameters; it's simply a matter of personal preference. For more information about setting parameters, see "Setting component parameters" on page 21.

To view information for a component instance in the Property inspector:

1 Select Window > Properties.

2 Select an instance of a component on the Stage.

3 To view parameters, click the Parameters tab.

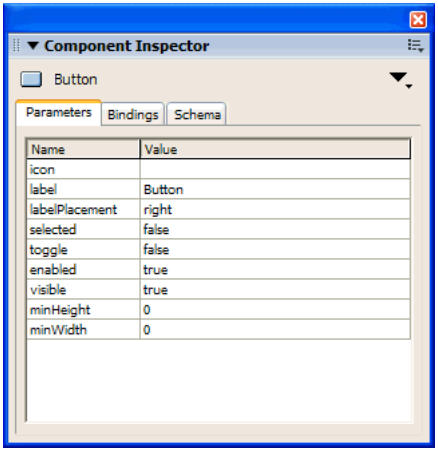


To view parameters for a component instance in the Component Inspector panel:

1 Select Window > Development Panels > Component Inspector.

2 Select an instance of a component on the Stage.

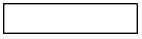
3 To view parameters, click the Parameters tab.



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, including 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 4, "Components Dictionary," on page 43. Components in Live Preview are not functional. To test component functionality, you can use the Control > Test Movie command.

A Button component with Live Preview enabled



A Button component with Live Preview disabled

To turn Live Preview on or off:

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

For more information, see Chapter 5, "Creating Components," on page 639.

Working with SWC files and compiled clips

Components included with Flash MX 2004 or Flash MX Professional 2004 are not FLA files—they are SWC files. SWC is the Macromedia file format for components. When you add a component to the Stage from the Components panel, a compiled clip symbol is added to the library. A SWC is a compiled clip that has been exported for distribution.

A movie clip can also be "compiled" in Flash and converted into a compiled clip symbol. The compiled clip symbol behaves just like the movie clip symbol from which it was compiled, but compiled clips display 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 in the Component Inspector panel and they include a live preview.

The components included with Flash MX 2004 or Flash MX Professional 2004 have already been turned into compiled clips. If you create a component, you may choose to export it as a SWC for distribution. For more information, see Chapter 5, "Creating Components," on page 639.

To compile a movie clip symbol:

- Select the movie clip in the library and right-click (Windows) or Control-click (Macintosh), and then select Convert to Compiled Clip.

To export a SWC:

- Select the movie clip in the library 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.

Adding components to Flash documents

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

- Beginning Flash users can use the Components panel to add components to Flash documents, specify basic parameters using the Property inspector or the Parameters tab in the Component Inspector panel, and use the `on()` event handler to control components.
- Intermediate Flash users can use the Components panel to add components to Flash documents and then use the Property inspector, ActionScript methods, or a combination of the two to specify parameters. They can use the `on()` event handler, or event listeners to handle component events.
- Advanced Flash programmers can use a combination of the Components panel and ActionScript to add components and specify properties, or choose to implement component instances at runtime using only ActionScript. They can use event listeners to control components.

If you edit the skins of a component and then add another version of the component, or a component that shares the same skins, you can choose to use the edited skins or replace the edited skins with a new set of default skins. If you replace the edited skins, all components using those skins are updated with default versions of the skins. For more information on how to edit skins, see Chapter 3, "Customizing Components," on page 27.

Adding components using the Components panel

After you add a component to a document using the Components panel, you can 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 panel.

To add a component to a Flash document 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 v2 components are SWCs) *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.

For more information, see "Setting component parameters" on page 21.

8 Change the size of the component as desired.

For more information on sizing specific component types, see the individual component entries in Chapter 4, "Components Dictionary," on page 43.

9 Change the color and text formatting of a component as desired, by doing one or more of the following:

- Set or change a specific style property value for a component instance using the `setStyle()` method available to all components. For more information, see `UIObject.setStyle()`.
- Edit multiple properties in the _global style declaration assigned to all v2 components.
- If desired, create a custom style declaration for specific component instances.

  For more information, see "Using styles to customize component color and text" on page 27.

10 Customize the appearance of the component if desired, by doing one of the following:

- Apply a theme (see "About themes" on page 34).
- Edit a component's skins (see "About skinning components" on page 36).

Adding components using ActionScript

To add a component to a document using ActionScript, you must first add it to the library.

You can use ActionScript methods to set additional parameters for dynamically added components. For more information, see Chapter 4, "Components Dictionary," on page 43.

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 from the Components panel to the Stage and delete it.

This adds the component to the library.

2 Select the frame in the Timeline where you want to place the component.

3 Open the Actions panel if it isn't already open.

4 Call the `createClassObject()` method to create the component instance at runtime.

This method can be called on its own, or from any component instance. It takes a component class name, an instance name for the new instance, a depth, and an optional initialization object as its parameters. You can specify the class package in the *`className`* parameter, as in the following:

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

Or you can import the class package, as in the following:

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

For more information, see `UIObject.createClassObject()`.

5 Use the ActionScript methods and properties of the component to specify additional options or override parameters set during authoring.

For detailed information on the ActionScript methods and properties available to each component, see their entries in Chapter 4, "Components Dictionary," on page 43.

About component label size and component width and height

If a component instance that has been added to a document is not large enough to display its label, the label text is clipped. If a component instance is larger than the text, the hit area extends beyond the label.

Use the Free Transform tool or the `setSize()` method to resize component instances. You can call the `setSize()` method from any component instance (see `UIObject.setSize()`). 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 remains the same. This may cause the component to be distorted in movie playback. For more information about sizing components, see their individual entries in Chapter 4, "Components Dictionary," on page 43.

Setting component parameters

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

All v2 components inherit properties and methods from the UIObject class and the UIComponent class; 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`), while the NumericStepper component has `nextValue` and `previousValue` properties (`NumericStepper.nextValue`, `NumericStepper.previousValue`).

Deleting components from Flash documents

To delete a component's instances from a Flash document, you delete the component from the library by deleting the compiled clip icon.

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 panel options menu.
3. In the Delete dialog box, click Delete to confirm the deletion.

Using code hints

When you are using ActionScript 2, you can strictly type 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, Flash displays a list of methods and properties available for CheckBox components, because you have typed the variable as a CheckBox. For more information on data typing, see “Strict data typing” in ActionScript Reference Guide Help. For information on using code hints when they appear, see “Using code hints” in ActionScript Reference Guide Help.

About component events

All components have events that are broadcast when the user interacts with a component or when something significant happens to the component. To handle an event, you write ActionScript code that executes when the event is triggered.

You can handle component events in the following ways:

- Use the `on()` component event handler.
- Use event listeners.

Using the on() event handler

The easiest way to handle a component event is to use the `on()` component 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.

When you use an `on()` event handler, an event object, `eventObj`, is automatically generated when the event is triggered and passed to the handler. The event object has properties that contain information about the event. The event object that is passed to the `on()` handler is always `eventObj`. For more information, see "UIEventDispatcher" on page 560.

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);
}
```

To use the on() event handler:

1 Drag a CheckBox component to the Stage from the Components panel.

2 Select the component and select Window > Actions.

3 In the Actions panel, enter the following code:

```
on(click){
  trace("The " + eventObj.type + " event was broadcast");
}
```

You can enter any code you wish between the curly braces(`{}`).

4 Select Control > Test Movie and select the check box to see the trace in the Output panel.

For more information, see each event entry in Chapter 4, "Components Dictionary," on page 43.

Using component event listeners

The most powerful way to handle component events is to use listeners. Events are broadcast by components and any object that is registered to the event broadcaster (component instance) as a listener can be notified of the event. The listener is assigned a function that handles the event. You can register multiple listeners to one component instance. You can also register one listener to multiple component instances.

To use the event listener model, you create a listener object with a property that is the name of the event. The property is assigned to a callback function. Then you call the `UIEventDispatcher.addEventListener()` method on the component instance that's broadcasting the event and pass it the name of the event and the name of the listener object. Calling the `UIEventDispatcher.addEventListener()` method is called "registering" or "subscribing" a listener, as in the following:

```
listenerObject.eventName = function(evtObj){
  // your code here
};
componentInstance.addEventListener("eventName", listenerObject);
```

In the above code, the keyword `this`, if used in the callback function, is scoped to the `listenerObject`.

The `evtObj` parameter is an event object that is automatically generated when an event is triggered and passed to the listener object callback function. The event object has properties that contain information about the event. For more information, see "UIEventDispatcher" on page 560.

For information about the events a component broadcasts, see each component's entry in Chapter 4, "Components Dictionary," on page 43.

To register an event listener, do the following:

1. Drag a Button component to the Stage from the Components panel.
2. In the Property inspector, enter the instance name **button**.
3. Drag a TextInput component to the Stage from the Components panel.
4. In the Property inspector, enter the instance name **myText**.
5. Select Frame 1 in the Timeline.
6. Select Window > Actions.
7. In the Actions panel, enter the following code:

   ```
   form = new Object();
   form.click = function(evt){
     myText.text = evt.target;
   }
   button.addEventListener("click", form);
   ```

   The `target` property of the event object is a reference to the instance broadcasting the event. This code displays the value of the `target` property in the text input field.

Additional event syntax

In addition to using a listener object, you can use a function as a listener. A listener is a function if it does not belong to an object. For example, the following code creates the listener function `myHandler` and registers it to `buttonInstance`:

```
function myHandler(eventObj){
  if (eventObj.type == "click"){
    // your code here
  }
}
buttonInstance.addEventListener("click", myHandler);
```

Note: In a function listener, the `this` keyword is `buttonInstance`, not the Timeline on which the function is defined.

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

```
myObj.handleEvent = function(o){
  if (o.type == "click"){
    // your code here
  } else if (o.type == "enter"){
    // your code here
  }
}
target.addEventListener("click", myObj);
target2.addEventListener("enter", myObj);
```

There is one additional event syntax style, which should be used only when you are authoring a component and know that a particular object is the only listener for an event. In such a situation, you can take advantage of the fact that the v2 event model always calls a method on the component instance that is the event name plus “Handler”. For example, if you want to handle the `click` event, you would write the following code:

```
componentInstance.clickHandler = function(o){
  // insert your code here
}
```

In the above code, the keyword `this`, if used in the callback function, is scoped to `componentInstance`.

For more information, see Chapter 5, “Creating Components,” on page 639.

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 focus. You don’t need to add a FocusManager instance to an application or write any code to activate the FocusManager.

If a RadioButton object receives focus, the FocusManager 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 FocusManager so the controls on that window become their own tab set, which prevents a user from inadvertently getting into 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 with a `tabIndex` property that is higher than the current value of `tabIndex`. Once the FocusManager class reaches the highest `tabIndex` property, it returns to zero. For example, in the following, 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 FocusManager uses the z-order. The z-order is set up primarily by the order components are dragged to the Stage, however, you can also use the Modify/Arrage/Bring-to-Front/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 name of the desired button, as in the following:

```
FocusManager.defaultPushButton = okButton;
```

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

Managing component depth in a document

If you want to position a component above or below another object in an application, you must use the DepthManager class. The DepthManager application programming interface (API) 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 windows float over content, and so on).

The DepthManager 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 DepthManager, call its methods (see "DepthManager class" on page 265).

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

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

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, you should 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 ProgressBar.

Upgrading version 1 components to version 2 architecture

The v2 components were written to comply with several web standards (regarding events, styles, getter/setter policies, and so on) and are very different from their v1 counterparts that were released with Macromedia Flash MX and in the DRKs that were released before Macromedia Flash MX 2004. V2 components have different APIs and were written in ActionScript 2.0. Therefore, using v1 and v2 components together in an application can cause unpredictable behavior. For information about upgrading v1 components to use version 2 event handling, styles, and getter/setter access to the properties instead of methods, see Chapter 5, "Creating Components," on page 639.

Flash applications that contain v1 components work properly in Flash Player 6 and Flash Player 7, when published for Flash Player 6 or Flash Player 6 release 65. If you would like 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 "Creating Classes with ActionScript 2.0" in ActionScript Reference Guide Help.

CHAPTER 3
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 Styles API.
- Apply a theme.
- Modify or replace a component's skins.

The Styles API (application programming interface) has methods and properties that allow you to change the color and text formatting of a component.

A theme is a collection of styles and skins that make up a component's appearance.

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.

Using styles to customize component color and text

Every component instance has style properties and `setStyle()` and `getStyle()` (see `UIObject.setStyle()` and `UIObject.getStyle()`) methods that you can use to modify and access style properties. 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.
- 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.
- Create custom style declarations and apply them to specific component instances.

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

- Create default class style declarations.

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

Changes made to style properties are not displayed when viewing components on the Stage using the Live Preview feature. For more information, see "Components in Live Preview" on page 17.

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. For example, the following code sets the text color on a Button instance called `myButton`:

```
myButton.setStyle("color", "0xFF00FF");
```

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. For more information, see "Setting style property values" on page 32.

Note: You should not call the UIObject.setStyle() method multiple times to set more than one property. If you want to change multiple properties, or change properties for multiple component instances, you should create a custom style format. For more information, see "Setting styles for specific components" on page 29.

To set or change a property for a single component instance:

1 Select the component instance on the Stage.

2 In the Property inspector, give it the instance name **myComp**.

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 blue:

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

The following syntax specifies a property and value for a component instance:

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

5 Select Control > Test Movie to view the changes.

For a list of supported styles, see "Supported styles" on page 33.

Setting global styles

The _global style declaration is assigned to all Flash components built with version 2 of the Macromedia Component Architecture (v2 components). The _global object has a property called `style` (`_global.style`) that is an instance of CSSStyleDeclaration. This `style` 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.

Some styles are set on a component class's CSSStyleDeclaration (for example, the `backgroundColor` style of the TextArea and TextInput components). Because the class style declaration takes precedence over the _global style declaration when determining style values, setting `backgroundColor` on the _global style declaration would have no effect on TextArea and TextInput. For more information, see "Using global, custom, and class styles in the same document" on page 30.

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 18.

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) the component appears.

4 Open the Actions panel.

5 Use the following syntax to change any properties on the _global style declaration. You only need to list the properties whose values you want to change, as in the following:

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

For a list of styles, see "Supported styles" on page 33.

6 Select Control > Test Movie to see the changes.

Setting styles for specific components

You can create custom style declarations to specify a unique set of properties for specific 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 to an instance. The CSSStyleDeclaration object is accessible if you have placed at least one component instance on the Stage.

You make changes to a custom style format in the same way that you edit the properties in the _global style declaration. Instead of the _global style declaration name, use the CSSStyleDeclaration instance. For more information on the _global style declaration, see "Setting global styles" on page 28.

For information about the properties of the CSSStyleDeclaration object, see "Supported styles" on page 33. For a list of which styles each component supports, see their individual entries in Chapter 4, "Components Dictionary," on page 43.

To create a custom style declaration for specific components:

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

For more information, see "Adding components to Flash documents" on page 18.

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) the component appears.

4 Open the Actions panel in expert mode.

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 style list:

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

Note: 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 `myStyle` style declaration:

```
styleObj.fontFamily = "_sans";
styleObj.fontSize = 14;
styleObj.fontWeight = "bold";
styleObj.textDecoration = "underline";
styleObj.color = 0x336699;
styleObj.setStyle("themeColor", "haloBlue");
```

9 In the same Script pane, use the following syntax to set the `styleName` property of two specific components to the custom style declaration:

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

You can also access styles on a custom style declaration using the `setStyle()` and `getStyle()` methods. The following code sets the `backgroundColor` style on the `newStyle` style declaration:

```
_global.styles.newStyle.setStyle("backgroundColor", "0xFFCCFF");
```

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, like TextArea and TextInput, have class style declarations predefined by default because their `borderStyle` and `backgroundColor` properties must be customized.

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

```
var o = _global.styles.CheckBox = new mx.styles.CSSStyleDeclaration();
o.color = 0x0000FF;
```

You can also access styles on a class style declaration using the `setStyle()` and `getStyle()` methods. The following code sets the color style on the RadioButton style declaration:

```
_global.styles.RadioButton.setStyle("color", "blue");
```

For more information on supported styles, see "Supported styles" on page 33.

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 _global style declaration, custom style declarations, style properties set directly on component instances, and default class style declarations. In such a situation, Flash determines the value of a property by looking for its definition in all these places in a specific order.

First, Flash looks for a style property on the component instance. If the style isn't set directly on the instance, Flash looks at the `styleName` property of the instance to see if a style declaration is assigned to it.

If the `styleName` property hasn't been assigned to a style declaration, Flash looks for the property on a default class style declaration. If there isn't a class style declaration, and the property doesn't inherit its value, the _global style declaration is checked. If the property is not defined on the _global style declaration, the property is `undefined`.

If there isn't a class style declaration, and the property does inherit its value, Flash looks for the property on the instance's parent. If the property isn't defined on the parent, Flash checks the parent's `styleName` property; if that isn't defined, Flash continues to look at parent instances until it reaches the _global level. If the property is not defined on the _global style declaration, the property is `undefined`.

The StyleManager tells Flash if a style inherits its value or not. For more information, see "StyleManager class" on page 502.

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

About color style properties

Color style properties behave differently than non-color 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. New color names can be added by using the StyleManager (see "StyleManager class" on page 502). 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

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

You can use any legal ActionScript identifier to create your own color names (for example, `"WindowText"` or `"ButtonText"`). Use the StyleManager to define new colors, as in the following:

```
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 4, "Components Dictionary," on page 43.

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. By setting the color style properties in the style declarations to ButtonText and WindowText and providing a user interface so the user can change the values of ButtonText and WindowText you can provide the same color schemes as Micosoft Windows, the Mac OS, or any operating system.

Setting style property values

You use the `UIObject.setStyle()` method to set a style property on a component instance, the global style declaration, a custom style declaration, or a class style declaration. The following code sets the `color` style of a radio button instance to red:

```
myRadioButton.setStyle("color", "red");
```

The following code sets the `color` style of the custom style declaration CheckBox:

```
_global.styles.CheckBox.setStyle("color", "white");
```

The `UIObject.setStyle()` method knows if a style is inheriting and notifies children of that instance if their style changes. It also notifies the component instance that it must redraw itself to reflect the new style. Therefore, you should use `setStyle()` to set or change styles. However, as an optimization when creating style declarations, you can directly set the properties on an object. For more information, see "Setting global styles" on page 28, "Setting styles for specific components" on page 29, and "Setting styles for a component class" on page 30.

You use the `UIObject.getStyle()` method to retrieve a style from a component instance, the global style declaration, a custom style declaration, or a class style declaration. The following code gets the value of the color property and assigns it to the variable `o`:

```
var o = myRadioButton.getStyle("color");
```

The following code gets the value of a style property defined on the _global style declaration:

```
var r = _global.style.getValue("marginRight");
```

If the style isn't defined, `getStyle()` may return the value `undefined`. However, `getStyle()` understands how style properties inherit. So, even though styles are properties, you should use `UIObject.getStyle()` to access them so you don't need to know whether the style is inheriting.

For more information, see `UIObject.getStyle()` and `UIObject.setStyle()`.

Supported styles

Flash MX 2004 and Flash MX Professional 2004 come with two themes: *Halo* (HaloTheme.fla) and *Sample* (SampleTheme.fla). Each theme supports a different set of styles. The Sample theme uses all the styles of the v2 styles mechanism and is provided so that you can see a sample of those styles in a document. The Halo theme supports a subset of the Sample theme styles.

The following style properties are supported by most v2 components in the Sample style. For information about which Halo styles are supported by individual components, see Chapter 4, "Components Dictionary," on page 43.

If any values other than allowed values are entered, the default value is used. This is important if you are re-using CSS style declarations that use values outside the Macromedia subset of values.

Components can support the following styles:

Style	Description
backgroundColor	The background of a component. This is the only color style that doesn't inherit its value. The default value is transparent.
borderColor	The black section of a three-dimensional border or the color section of a two-dimensional border. The default value is 0x000000 (black).
borderStyle	The component border: either "none", "inset", "outset", or "solid". This style does not inherit its value. The default value is "solid".
buttonColor	The face of a button and a section of the three-dimensional border. The default value is 0xEFEEEF (light gray).
color	The text of a component label. The default value is 0x000000 (black).
disabledColor	The disabled color for text. The default color is 0x848384 (dark gray).
fontFamily	The font name for text. The default value is _sans.
fontSize	The point size for the font. The default value is 10.
fontStyle	The font style: either "normal" or "italic". The default value is "normal".
fontWeight	The font weight: either "normal" or "bold". The default value is "normal".
highlightColor	A section of the three-dimensional border. The default value is 0xFFFFFF (white).
marginLeft	A number indicating the left margin for text. The default value is 0.
marginRight	A number indicating the right margin for text. The default value is 0.
scrollTrackColor	The scroll track for a scroll bar. The default value is 0xEFEEEF (light gray).

Style	Description
shadowColor	A section of the three-dimensional border. The default value is 0x848384 (dark gray).
symbolBackgroundColor	The background color of check boxes and radio buttons. The default value is 0xFFFFFF (white).
symbolBackgroundDisabledColor	The background color of check boxes and radio buttons when disabled. The default value is 0xEFEEEF (light gray).
symbolBackgroundPressedColor	The background color of check boxes and radio buttons when pressed. The default value is 0xFFFFFF (white).
symbolColor	The check mark of a check box or the dot of a radio button. The default value is 0x000000 (black).
symbolDisabledColor	The disabled check mark or radio button dot color. The default value is 0x848384 (dark gray).
textAlign	The text alignment: either "left", "right", or "center". The default value is "left".
textDecoration	The text decoration: either "none" or "underline". The default value is "none".
textIndent	A number indicating the text indent. The default value is 0.

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 was developed to let you provide a responsive, expressive experience for your users. Flash MX 2004 and Flash MX Professional 2004 include one additional theme called Sample (SampleTheme.fla). The Sample theme allows you to experiment with the full set of styles available to v2 components. (The Halo theme uses only a subset of the available styles.) The theme files are located in the following folders:

- First Run\ComponentFLA (Windows)
- First Run/ComponentFLA (Macintosh)

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 a two-dimensional theme and a three-dimensional theme.

The v2 components use skins (graphic or movie clip symbols) to display their visual 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 styles. You must write ActionScript code to create a global style declaration and any additional style declarations. For more information, see "Using styles to customize component color and text" on page 27.

Applying a theme to a document

To apply a new theme to a document, open a theme FLA 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 v2 components in Flash, or select File > New and create a new document that uses v2 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, located in the Flash 2004/en/Configuration/SampleFLA folder.
4. In the theme's Library panel, select Flash UI Components 2 > Themes > MMDefault and drag the Assets folder of any component(s) in your document to the ThemeApply.fla library.

   If you're unsure about which components are in the documents, you can drag the entire Themes folder to the Stage. The skins inside the Themes folder in the library 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. Select Control > Test Movie to see the document with the new theme applied.

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.

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 like **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 Themes > MMDefault > *Component* Assets folder (in this example, Themes > MMDefault > 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).

6 When you have finished 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 in the previous section, "Applying a theme to a document" on page 35.

About skinning components

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

Macromedia v2 components are compiled clips—you cannot see their assets in the library. However, FLA files are installed with Flash 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 allows you to drag a theme onto the Stage in a document to change its appearance. For more information about themes, see "About themes" on page 34. 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.

Each component is composed of many skins. For example, the down arrow of the ScrollBar subcomponent is made up of three skins: ScrollDownArrowDisabled, ScrollDownArrowUp, and ScrollDownArrowDown. Some components share skins. Components that use scroll bars—including 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 a component.

The .as file that defines each component class contains code that loads specific skins for the component. Each component skin has 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 has the skin property name `downArrowDownName`. The default value of the `downArrowDownName` property is `"ScrollDownArrowDown"`, which is the Linkage Identifier of the skin symbol. You can edit skins and apply them to a component by using these skin properties. 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.

Choose one of the following ways to skin a component based on what you want to do:

- 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 34).

  Note: This method of skinning is recommended for beginners because it doesn't require any scripting.

- To use different skins for multiple instances of the same component, edit the existing skins and set skin properties (see the next section, "Editing component skins" on page 37, and "Applying an edited skin to a component" on page 38).
- To change skins in a subcomponent (such as a scroll bar in a List component), subclass the component (see "Applying an edited skin to a subcomponent" on page 39).
- 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 41).

Note: The above methods are listed from top to bottom according to ease of use.

Editing 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 edit a skin, do the following:

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 skin or 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 on the symbol), and give the symbol a unique name like MyRadioTrueUp.
5. Select the Advanced button in the Symbol Properties dialog 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 have finished 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 the next section, "Applying an edited skin to a component" on page 38, "Applying an edited skin to a subcomponent" on page 39, or "Changing skin properties in the prototype" on page 41. For information about how to apply a new skin, see "About skinning components" on page 36.

Note: Changes made to component skins are not displayed when viewing components on the Stage using Live Preview.

Applying an edited skin to a component

Once you have edited a skin, you must apply it to a component in a document. You can either 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 an edited skin, do the following:

1. Select File > New to create a new Flash document.
2. Select File > Save and give it 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 an edited skin, do the following:

1. Select File > New to create a new Flash document.
2. Select File > Save and give it 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 an edited skin 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 allows you to 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 4, "Components Dictionary," on page 43.

To apply a new skin to a subcomponent, do the following:

1 Follow the steps in "Editing component skins" on page 37, 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 it 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 component to the Library panel, but doesn't make the component 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 either 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: You must set enough data to have the scroll bars show up, 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. For more information about the prototype object, see `Function.prototype` in ActionScript Dictionary Help.

To use #initclip to apply an edited skin to all components in a document, do the following:

1 Follow the steps in "Editing component skins" on page 37, 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 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 Lists; any TextArea or ScrollPane scroll bars would not be skinned.

To use #initclip to apply an edited skin to specific components in a document, do the following:

1 Follow the steps in "Editing component skins" on page 37, 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 it a unique name like **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 it takes 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, select the Publish Settings button.

15 Select the ActionScript version Settings button.

16 Click the Plus (+) button to add a new classpath, and select the Target button to browse to the location of the MyComboBox.as file on your hard drive.

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 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 4, "Components Dictionary," on page 43.

To skin a subcomponent, do the following:

1 Follow the steps in "Editing component skins" on page 37, 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 it 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 actions of MyComboBox:

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

8 Drag a ComboBox component to the Stage.

9 In the Property inspector, enter as many Label parameters as it takes for the vertical scroll bar to appear.

10 Select File > Save.

11 Select File > New and create a new ActionScript file (Flash Professional only).

12 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;
  }
}
```

13 Select File > Save and save this file as **MyComboBox.as**.

14 Click a blank area on the Stage and, in the Property inspector, select the Publish Settings button.

15 Select the ActionScript version Settings button.

16 Click the Plus (+) button to add a new classpath, and select the Target button to browse to the location of the MyComboBox.as file on your hard drive.

17 Select Control > Test Movie.

CHAPTER 4
Components Dictionary

This reference chapter describes each component and each component's 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 following tables:

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 the user with a question and buttons to capture their response.
Button component	A resizable button that can be customized with a custom icon.
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.
DateChooser component (Flash Professional only)	Allows users to select a date or dates from a calendar.
DateField component (Flash Professional only)	A unselectable text field with a calendar icon. When a user clicks anywhere inside the bounding box of the component, a DateChooser component is displayed.

Component	Description
DataGrid component (Flash Professional only)	Allows users to display and manipulate multiple columns of data.
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)	Allows users to select one command from a list; a standard desktop application menu.
MenuBar component (Flash Professional only)	A horizontal bar of menus.
NumericStepper component	Clickable arrows that raise and lower the value of an number.
ProgressBar component	Displays the progress of a process, usually loading.
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 that display content to the user.

Data components

Component	Description
Data binding classes (Flash Professional only)	These classes implement the Flash runtime data binding functionality.
DataHolder component (Flash Professional only)	Holds data and can be used as a connector between components.
DataProvider API	This component is the model for linear-access lists of data. This model provides simple array-manipulation capabilities that broadcast their changes.
DataSet component (Flash Professional only)	A building block for creating data-driven applications.
RDBMSResolver component (Flash Professional only)	Allows you to save data back to any supported data source. This resolver 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)	These classes allow access to web services that use Simple Object Access Protocol (SOPAP) found in the mx.services package.
WebServiceConnector class (Flash Professional only)	Provides scriptless access to web service method calls.

Component	Description
XMLConnector component (Flash Professional only)	Reads and writes XML documents using the HTTP GET and POST methods.
XUpdateResolver component (Flash Professional only)	Allows you to save data back to any supported data source. This resolver component translates the DeltaPacket 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 325.

Managers

Component	Description
DepthManager class	Manages the stacking depths of objects.
FocusManager class	Handles Tab key navigation between components on the screen. Also handles focus changes as users click in the application.
PopUpManager class	Allows you to create and delete pop-up windows.
StyleManager class	Allows you to register styles and manages inherited styles.

Screens

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

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 a subclass of the UIObject class (which includes all components and screens built using version 2 of the Macromedia Component Architecture), but most commonly 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 press to navigate between the accordion's children. An accordion has a vertical layout with header buttons that span the width of the component. There is one header 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 wraps from last to first without changing the selected child.
Up arrow, Left arrow	Moves focus to the previous child header. Focus wraps 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 wraps from the last child to the first child.
Page Up	Selects the previous child. Selection wraps 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 will never be 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 will never be another header in the same accordion.

The Accordion component cannot be made accessible to screen readers.

Using the Accordion component (Flash Professional only)

The Accordion component can be used to present multi-part 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

The following are authoring parameters that you can set for each Accordion component instance in the Property inspector or in the Component Inspector panel:

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

childNames An array specifying the instance names of the accordion's children. The default value is [] (empty array).

childLabels An array specifying the text labels to use on the accordion's headers. The default value is [] (empty array).

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

You can write ActionScript to control these and additional options for the Accordion component using its properties, methods, and events. For more information, see "Accordion class (Flash Professional only)" on page 50.

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 users enter their shipping address, 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 show up in the library, the addressForm screen is a symbol of the Screen class (which is a subclass of the View class), which an accordion can use as a child.

3 With the form selected, in the Property inspector, set its 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 pane of the Component Inspector panel.

Position the form elements in the upper left corner of the form. The upper left 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 form 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 have identical components.

- 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, do the following:

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, 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 pane of the Component Inspector panel.

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, do the following:

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 it 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 4, 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 both 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 `setSize()` method and the Transform tool only change 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 non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

An Accordion component supports the following Halo styles:

Style	Description
`themeColor`	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
`backgroundColor`	The background color.
`borderColor`	The border color.
`borderStyle`	The border style; possible values are "none", "solid", "inset", "outset", "default", "alert". The "default" value is the look of the Window component's border and the "alert" value is the look of the Alert component's border.

Style	Description
headerHeight	The height of the header buttons in pixels.
color	The header text color.
disabledColor	The color of a disabled accordion.
fontFamily	The font name for the header labels.
fontSize	The point size for the font of the header labels.
fontStyle	The font style for the header labels; either "normal", or "italic".
fontWeight	The font weight for the header labels; either "normal", or "bold".
textDecoration	The text decoration; either "none", or "underline".
openDuration	The duration, in milliseconds, of the transition animation.
openEasing	The tweening function used by the animation.

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 36.

An Accordion component is composed of its border and background, its header buttons, and its children. The border and background are styleable, but not skinnable. The headers are skinnable, but not styleable, using the subset of skins inherited from button listed below. An Accordion component uses the following skin properties to dynamically skin the header buttons:

Property	Description	Default value
falseUpSkin	The up state.	accordionHeaderSkin
falseDownSkin	The pressed state.	accordionHeaderSkin
falseOverSkin	The rolled-over state.	accordionHeaderSkin
trueUpSkin	The toggled state.	accordionHeaderSkin

Accordion class (Flash Professional only)

Inheritance UIObject > UIComponent > View > Accordion

ActionScript Class Name mx.containers.Accordion

An Accordion is a component that 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 panel.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.Accordion.version);
```

Note: The following code returns undefined: `trace(myAccordionInstance.version);`.

Method summary for 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.

Inherits all methods from UIObject, UIComponent and mx.core.View.

Property summary for 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.

Inherits all properties from UIObject, UIComponent and mx.core.View.

Event summary for the Accordion class

Event	Description
`Accordion.change`	Broadcast to all registered listeners when the `selectedIndex` and `selectedChild` properties of an accordion change due to a user's mouse click or key press.

Inherits all events from UIObject, UIComponent and mx.core.View.

Accordion.change

Availability

Flash Player 6 version 79.

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 key press changes the value `selectedChild` or `selectedIndex`—not when the value is changed with ActionScript. This event is broadcast before the transition animation occurs.

V2 components use a dispatcher/listener event model. The Accordion component dispatches a `change` event when one of its buttons is pressed 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 a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

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 `change` handler in the `evtObject` parameter. When the `change` event is broadcast, a `trace` statement is sent to the Output panel, as follows:

```
myAccordionListener = new Object();
myAccordionListener.change = function(){
  trace("Changed to different view");
}
myAccordion.addEventListener("change", myAccordionListener);
```

Accordion.createChild()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.createChild(classOrSymbolName, instanceName[, initialProperties])
```

Parameters

classOrSymbolName This parameter can either be 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 a View 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` This string specifies the text label that the new child instance uses on its header.
- `icon` This string 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 movie clip symbol PaymentForm 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 version 79.

Edition

Flash MX Professional 2004.

```
myAccordion.createSegment(classOrSymbolName, instanceName[, label[, icon]])
```

Parameters

classOrSymbolName This parameter can be 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 a View or a subclass of View.

instanceName The instance name of the new instance.

label This string specifies the text label that the new child instance uses on its header. This parameter is optional.

icon This string is a 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 version 79.

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 that 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 the `destroyChildAt()` method 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 version 79.

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 that 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.numChildren
```

Description

Property (inherited from View); indicates the number of children (child UIObjects) 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 7 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myAccordion.selectedChild
```

Description

Property; the selected child if one or more children exist; undefined if no children exist. This property is either of type UIObject, or undefined.

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 gets 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 version 79.

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.selectedChild`, `Accordion.numChildren`

Alert component (Flash Professional only)

The Alert component allows you to pop up a window that presents the user with a message and response buttons. The Alert 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 the following buttons: Yes, No, OK, and Cancel. You can change the text labels on the buttons by using the following properties: `Alert.yesLabel`, `Alert.noLabel`, `Alert.okLabel`, and `Alert.cancelLabel`. You cannot change the order of the buttons in an Alert window; the button order is always OK, Yes, No, Cancel.

To pop up an Alert window, you must call the `Alert.show()` method. In order to call the method successfully, the Alert component must be in the library. You must drag the Alert component from the Components panel to the Stage and then delete the Alert component from the Stage. This adds the component to the Library but doesn't make it visible in the document.

The live preview for the Alert component is an empty window.

The text and buttons of an Alert window can be made accessible to screen readers. When you add the Alert 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.AlertAccImpl.enableAccessibility();
```

You only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

Using the Alert component (Flash Professional only)

The Alert can be used whenever you want to announce something to a user. For example, you could pop up an Alert when a user doesn't fill out a form properly, or when a stock hits a certain price, or when a user quits an application without saving his session.

Alert parameters

There are no authoring parameters for the Alert component. You must call the ActionScript `Alert.show()` method to pop up 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 61.

Creating an application with the Alert component

The following procedure explains how to add a Alert component to an application while authoring. In this example, the Alert component pops up when a stock hits a certain price.

To create an application with the Alert component, do the following:

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 either button is pressed, the `myClickHandler` function is called. But when the OK button is pressed, the `startStockApplication()` method is called.
4. Control > Test Movie.

Customizing the Alert component (Flash Professional only)

The Alert positions itself in the center of the component that was passed as its *parent* parameter. The parent must be a UIComponent. 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 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 non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

An Alert component supports the following Halo styles:

Style	Description
themeColor	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
color	The text of a component label.
disabledColor	The disabled color for text.
fontFamily	The font name for text.
fontSize	The point size for the font.
fontStyle	The font style; either "normal", or "italic".
fontWeight	The font weight; either "normal", or "bold".
textDecoration	The text decoration; either "none", or "underline".
buttonStyleDeclaration	A class (static) CSSStyleDeclaration for the button's text styles.
messageStyleDeclaration	A class (static) CSSStyleDeclaration for the message's text, border, and background styles.
titleStyleDeclaration	A class (static) CSSStyleDeclaration for the title's text styles.

Using skins with the Alert component

The Alert component uses the Window skins to represent the visual states of its buttons and title bar. To skin the buttons and title bar while authoring, modify skin symbols in the Flash UI Components 2/Themes/MMDefault/Window Assets skins states folder in the library of one of the themes FLA files. For more information, see "About skinning components" on page 36.

There is ActionScript code in the RectBorder.as class that the Alert component uses to draw its borders. You can use RectBorder styles to modify an Alert component as follows:

```
var myAlert = Alert.show("This is a test of errors", "Error", Alert.OK |
  Alert.CANCEL, this);
myAlert.setStyle("borderStyle", "inset");
```

For information about RectBorder styles, see "Using skins with the List component" on page 290.

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 button.	ButtonSkin
buttonDown	The pressed state of the button.	ButtonSkin
buttonOver	The rolled-over state of button.	ButtonSkin
titleBackground	The window title bar.	TitleBackground

Alert class (Flash Professional only)

Inheritance UIObject > UIComponent > View > ScrollView > Window > 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 pop up 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 after the call to `Alert.show()` run right away. You must add listeners to handle the `click` events that are broadcast when a user presses 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()` would not return until the user has taken an action, such as pushing a button.

Method summary for the Alert class

Event	Description
`Alert.show()`	Creates an Alert window with optional parameters.

Inherits all methods from UIObject and UIComponent.

Property summary for 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.cancelLabel`	The label text for the Cancel button.
`Alert.noLabel`	The label text for the No button.
`Alert.okLabel`	The label text for the OK button.
`Alert.yesLabel`	The label text for the Yes button.

Inherits all properties from UIObject and UIComponent.

Event summary for the Alert class

Event	Description
`Alert.click`	Broadcast when a button in an Alert window is clicked.

Inherits all events from UIObject and UIComponent.

Alert.buttonHeight

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
Alert.buttonHeight
```

Description

Property (class); a class property (static) that changes the height of the buttons.

See also

Alert.buttonWidth

Alert.buttonWidth

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
Alert.buttonWidth
```

Description

Property (class); a class property (static) that changes the width of the buttons.

See also

Alert.buttonHeight

Alert.click

Availability

Flash Player 6 version 79.

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.

V2 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 is triggered, it automatically passes an event object (*`eventObject`*) to the handler. Each event object has a set of 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 one of the following depending on which button was clicked: `Alert.OK`, `Alert.CANCEL`, `Alert.YES`, `Alert.NO`. For more information about event objects, see "Event Objects" on page 562.

Example

In the following example, a handler called `myClickHandler` is defined and passed to the `Alert.show()` method as the 5th parameter. The event object is captured by `myClickHandler` in the `evt` parameter. The `detail` property of the event object is then used within a `trace` statement to send the name of the button that was clicked (`Alert.OK` or `Alert.CANCEL`) to the Output panel, as follows:

```
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.cancelLabel

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
Alert.cancelLabel
```

Description

Property (class); a class property (static) 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.noLabel

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
Alert.noLabel
```

Description

Property (class); a class property (static) 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.okLabel

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
Alert.okLabel
```

Description

Property (class); a class property (static) 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 version 79.

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 the *title* parameter is not specified, the title bar is blank.

flags An optional parameter that indicates the button or 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. The value can be one or more of the following:

- `Alert.OK`
- `Alert.CANCEL`
- `Alert.YES`
- `Alert.NO`

You can also use `Alert.NONMODAL` to indicate that the Alert window is non-modal. A non-modal 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 value of the button flag that was clicked (`Alert.OK`, `Alert.CANCEL`, `Alert.YES`, `Alert.NO`). This handler may be a function or an object. For more information, see Chapter 2, "Using component event listeners," on page 22.

icon A string that is the linkage identifier of a symbol in the library to use as an icon that is displayed to the left of the text. This parameter is optional.

defaultButton Indicates which button is clicked when a user presses Enter (Windows) or Return (Macintosh). This parameter can be one of the following values:

- `Alert.OK`
- `Alert.CANCEL`
- `Alert.YES`
- `Alert.NO`

Returns

The instance of the Alert class 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 aligned to the left. The icon appears to the left of the message text. The buttons appear 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:

```
Alert.show("Hello, world!");
```

The following code defines a click handler that sends a message to the Output panel about which button was clicked:

```
myClickHandler = function(evt){
  trace (evt.detail + "was clicked");
}
Alert.show("This is a test of errors", "Error", Alert.OK | Alert.CANCEL, this,
  myClickHandler);
```

Note: The event object's detail property returns a number to represent each button. The OK buttons is 4, the cancel button is 8, the yes button is 1, and the no button is 2.

Alert.yesLabel

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
Alert.yesLabel
```

Description

Property (class); a class property (static) 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 24 or "FocusManager class" on page 270.

A live preview of each Button instance reflects changes made to parameters in the Property inspector or Component Inspector panel while 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 to enable accessibility for the Button component:

```
mx.accessibility.ButtonAccImpl.enableAccessibility();
```

You only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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 from the Options menu, and enter a linkage identifier. This is the value to enter for the icon parameter in the Property inspector or Component Inspector panel. You can also enter this value for the Button.icon ActionScript property.

Note: If an icon is larger than the button it will extend beyond the button's borders.

Button parameters

The following are authoring parameters that you can set for each Button component instance in the Property inspector or in the Component Inspector panel:

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 pressed and returns to the up state when pressed 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 Button components using its properties, methods, and events. For more information, see Button class.

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 will open a Help system when a user presses it.

To create an application with the Button component, do the following:

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:

```
clippyListener = new Object();
clippyListener.click = function (evt){
  clippyHelper.enabled = evt.target.selected;
}
helpBtn.addEventListener("click", clippyListener);
```

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 both 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). 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 clips to fit.

If an icon is larger than the button it will extend 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 non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

A Button component supports the following Halo styles:

Style	Description
`themeColor`	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.

Style	Description
fontStyle	The font style: either "normal", or "italic".
fontWeight	The font weight: either "normal", or "bold".

Using skins with the Button component

The Button component uses the ActionScript drawing API to draw the button states. To skin the Button component while authoring, modify the ActionScript code within the ButtonSkin.as file located in the First Run\Classes\mx\skins\halo folder.

If you use the UIObject.createClassObject() method to create a Button component instance dynamically (at runtime), you can skin it dynamically. To skin a component at runtime, set the skin properties of the initObject parameter that is passed to the createClassObject() method. These skin properties set the names of the symbols to use as the button's states, both with and without an icon.

If you set the icon parameter while authoring or the icon ActionScript property at runtime, the same linkage identifier is assigned to three icon states: falseUpIcon, falseDownIcon, and trueUpIcon. If you want to designate a unique icon for any of the eight icon states (if, for example, you want a different icon to appear when a user presses a button) you must set properties of the initObject parameter that is passed to the createClassObject() method.

The following code creates an object called initObject to use as the initObject parameter and sets skin properties to new symbol linkage identifiers. The last line of code calls the createClassObject() method to create a new instance of the Button class with the properties passed in the initObject parameter, as follows:

```
var initObject = new Object();
initObject.falseUpIcon = "MyFalseUpIcon";
initObject.falseDownIcon = "MyFalseDownIcon";
initObject.trueUpIcon = "MyTrueUpIcon";
createClassObject(mx.controls.Button, "ButtonInstance", 0, initObject);
```

For more information, see "About skinning components" on page 36, and UIObject.createClassObject().

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 clicked. 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 uses the following skin properties:

Property	Description
falseUpSkin	The up state. The default value is RectBorder.
falseDownSkin	The pressed state. The default value is RectBorder.
falseOverSkin	The over state. The default value is RectBorder.
falseDisabledSkin	The disabled state. The default value is RectBorder.

Property	Description
`trueUpSkin`	The toggled state. The default value is RectBorder.
`trueDownSkin`	The pressed-toggled state. The default value is RectBorder.
`trueOverSkin`	The over-toggled state. The default value is RectBorder.
`trueDisabledSkin`	The disabled-toggled state. The default value is RectBorder.
`falseUpIcon`	The icon up state. The default value is undefined.
`falseDownIcon`	The icon pressed state. The default value is undefined.
`falseOverIcon`	The icon over state. The default value is undefined.
`falseDisabledIcon`	The icon disabled state. The default value is undefined.
`trueUpIcon`	The icon toggled state. The default value is undefined.
`trueOverIcon`	The icon over-toggled state. The default value is undefined.
`trueDownIcon`	The icon pressed-toggled state. The default value is undefined.
`trueDisabledIcon`	The icon disabled-toggled state. The default value is undefined.

Button class

Inheritance UIObject > UIComponent > SimpleButton > Button

ActionScript Class Name mx.controls.Button

The properties of the Button class allow you to add an icon to a button, create a text label, or indicate whether the button acts as a push button, or a toggle switch at runtime.

Setting a property of the Button class with ActionScript overrides the parameter of the same name set in the Property inspector or Component Inspector panel.

The Button component uses the FocusManager 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 24.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.Button.version);
```

Note: The following code returns undefined: `trace(myButtonInstance.version);`.

The Button component class is different from the ActionScript built-in Button object.

Method summary for the Button class

Inherits all methods from UIObject and UIComponent.

Property summary for the Button class

Method	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.
Button.icon	Specifies an icon for a button instance.
Button.label	Specifies the text that appears within a button.
Button.labelPlacement	Specifies the orientation of the label text in relation to an icon.
Button.selected	When the toggle property is true, specifies whether the button is pressed (true) or not (false).
Button.toggle	Indicates whether the button behaves as a toggle switch.

Inherits all properties from UIObject and UIComponent.

Event summary for the Button class

Method	Description
Button.click	Broadcast when the mouse is pressed over a button instance or when the Spacebar is pressed.

Inherits all events from UIObject and UIComponent.

Button.click

Availability

Flash Player 6 version 79.

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);
}
```

Please note that this differs from the behavior of `this` when used inside an `on()` handler attached to a regular Flash button symbol. When `this` is used inside an `on()` handler attached to a button symbol, it 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 is triggered. When the event is triggered, it automatically passes an event object (*`eventObject`*) to the listener object method. The event object has a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `addEventListener()` method (See `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 of code labels the button. The second line specifies that the button act like a toggle switch. The third line creates a listener object called `form`. The fourth line defines a function for the `click` event on the listener object. Inside the function is a trace action 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 `Button.selected` property is accessed from the event object's `target` property. The last line calls the `addEventListener()` method from `buttonInstance` and passes it the `click` event and the `form` listener object as parameters, as in the following:

```
buttonInstance.label = "Click Test"
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`, as in the following:

```
on(click){
  trace("button component was clicked");
}
```

See also

`UIEventDispatcher.addEventListener()`

SimpleButton.emphasized

Availability

Flash Player 6 version 79.

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 looks if a default push button. In general, use the `FocusManager.defaultPushButton` property instead of setting the `emphasized` property directly. The default value is `false`.

The `emphasized` property is a static property of the SimpleButton class. Therefore, you must access it directly from SimpleButton, as in the following:

```
SimpleButton.emphasizedStyleDeclaration = "foo";
```

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 = "foo";
myButton.emphasized = true;
```

See also

`SimpleButton.emphasizedStyleDeclaration`

SimpleButton.emphasizedStyleDeclaration

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

buttonInstance.emphasizedStyleDeclataion

Description

Property; a string indicating the style declaration that formats a button when the `emphasized` property is set to `true`.

See also

`SimpleButton.emphasized`

Button.icon

Availability

Flash Player 6 version 79.

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 will resize automatically. If an icon is larger than a button, the icon will extend 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 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.

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 version 79.

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 panel. 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 version 79.

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 placed below 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);
```

Button.selected

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`buttonInstance``.selected`

Description

Property; a Boolean value specifying whether a button is pressed (`true`) or not (`false`). The value of the `toggle` property must be `true` to set the `selected` property to `true`. If the `toggle` property is `false`, assigning a value of `true` to the `selected` property has no effect. The default value is `false`.

The `click` event is not triggered when the value of the `selected` property changes with ActionScript. It is triggered when a user interacts with the button.

Example

In the following example, the `toggle` property is set to `true` and the `selected` property is set to `true` which puts the button in a pressed state. The `trace` action sends the value `true` to the Output panel:

```
ButtonInstance.toggle = true; // toggle needs to be true in order to set the
  selected property
ButtonInstance.selected = true; //displays the toggled state of the button
trace(ButtonInstance.selected); //traces- true
```

See also

`Button.toggle`

Button.toggle

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`buttonInstance``.toggle`

Description

Property; a Boolean value specifying whether a button acts like a toggle switch (`true`) or a push button (`false`); the default value is `false`. When a toggle switch is pressed, it stays in a pressed state until it's clicked again.

Example

The following code sets the `toggle` property to `true`, which makes the `myButton` instance behave like a toggle switch:

```
myButton.toggle = true;
```

CellRenderer API

The CellRenderer API is a set of properties and methods that the List-based components (List, DataGrid, Tree, and Menu) 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, or any class you create.

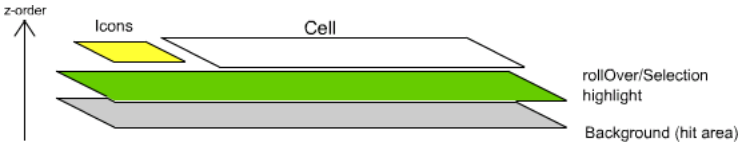
Understanding the List class

To use the CellRenderer API it is important to have an advanced understanding of the List class. The DataGrid, Tree, and Menu components are extension of the List class, so understanding the List class allows you to understand them as well.

Note: A component is a class but a class isn't necessarily a component.

About the composition of the List class

List classes 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, many cells). In the default case, these cells are TextField objects that implement the CellRenderer API. However, you can tell a List 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 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 class

List classes use a fairly complex algorithm to scroll. 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 by setting the value of the old row to the new item in the view and moving 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. Because rows are recycled, it is the responsibility of the cell renderer to 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()`, `CellRenderer.setValue()`) that the List-based component uses to communicate with the cell.

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, say a cell has a check box within it that causes a row to be selected when it's clicked. The cell renderer needs a reference to the List-based component that contains it in order to call the `selectedIndex` property of the List-based component. 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 APIs; the cell receives them automatically when it is placed inside the List-based component.

Methods to implement for the CellRenderer API

You must write a class with the following methods so that the List, DataGrid, Tree, or Menu, can communicate with the cell:

Name	Description
`CellRenderer.getPreferredHeight()`	Returns the preferred height of a cell.
`CellRenderer.getPreferredWidth()`	Returns the preferred width of a cell.
`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 following are the methods that the List, DataGrid, Tree, and Menu give to the cell when it is created within the component. You do not need to implement these methods.

Name	Description
`CellRenderer.getDataLabel()`	Returns a string containing the name of the cell renderer's data field.
`CellRenderer.getCellIndex()`	Returns an object with two fields, `columnIndex` and `rowIndex`, that indicate the position of the cell.

Properties provided by the CellRenderer API

The following is the property that the List, DataGrid, Tree, and Menu give to the cell when it is created within the component. You do not need to implement this property.

Name	Description
`CellRenderer.listOwner`	A reference to the List that contains the cell.

CellRenderer.getDataLabel()

Availability

Flash Player 6 version 79.

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.

Example

The following code helps the cell discover that it's rendering the data field `"Price"`. The variable `p` is now equal to "Price":

```
var p = getDataLabel();
```

CellRenderer.getCellIndex()

Availability

Flash Player 6 version 79.

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 grid. Each field is an integer that indicates a cell's column position and item position. For any components other than the DataGrid, the value of `columnIndex` is always 0.

Example

This example edits a DataGrid's dataProvider from within a cell:

```
var index = getCellIndex();
var colName = listOwner.getColumnAt(index.columnIndex).columnName;
listOwner.dataProvider.editField(index.itemIndex, colName, someVal);
```

CellRenderer.getPreferredHeight()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

componentInstance.getPreferredHeight()

Parameters

None.

Returns

The correct height for the cell.

Description

Method; 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.

Example

This example returns the value 20, which indicates that the cell wants to be 20 pixels high:

```
function getPreferredHeight(Void) :Number
{
   return 20;
}
```

CellRenderer.getPreferredWidth()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

componentInstance.getPreferredWidth()

Parameters

None.

Returns

Nothing.

Description

Method; the preferred width of a cell. If you specify more width than the component has, the cell may be cut off.

Example

This example returns the value 3, which indicates that the cell wants to be three times as big as the length of the string it is rendering:

```
function getPreferredHeight(Void) : Number
{
   return myString.length*3;
}
```

CellRenderer.listOwner

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

componentInstance.listOwnner

Description

Property; a reference to the list that owns the cell. That list can be a DataGrid, Tree, or List.

Example

This example finds the list's selected item in a cell:

```
var s = listOwner.selectedItem;
```

CellRenderer.setSize()

Availability

Flash Player 6 version 79.

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; allows the list to tell its cells at what size they should lay themselves out. The CellRenderer should do layout so that it fits within the area described, or visual display from the cell may bleed into other parts of the list and appear broken.

Example

This example sizes an image within the cell to fit within the bounds specified by the list:

```
function setSize(w:Number, h:Number) : Void
{
   image._width = w-2;
   image._height = w-2;
   image._x = image._y = 1;
}
```

CellRenderer.setValue()

Availability

Flash Player 6 version 79.

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 any properties of this object it wants for rendering.

selected A Boolean value that indicates whether the row the cell is on is selected (`true`) or not (`false`).

Returns

Nothing.

Description

Method; takes the values given and creates a representation of them within the cell. This clears up any difference in what was displayed in the cell and what needs to be displayed in the cell for the new item. It is important to remember that any cell could display many values during its time in the list. This is the most important method in any cell renderer.

Example

This 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;
}
```

CheckBox component

A check box is a square box that can be either selected or deselected. When it is selected, a check 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 checkbox 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 24 or "FocusManager class" on page 270.

A live preview of each CheckBox instance reflects changes made to parameters in the Property inspector or Component Inspector panel while 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 only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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

The following are authoring parameters that you can set for each CheckBox component instance in the Property inspector or in the Component Inspector panel:

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 CheckBox components using its properties, methods, and events. For more information, see CheckBox class.

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" permitting the customer to restrict his or her 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 that are enabled only when "Restrict Age" is selected.

To create an application with the CheckBox component, do the following:

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, that have already been placed on Stage. For more information about the `click` event, see `CheckBox.click`. For more information about the TextInput component, see "TextInput component" on page 516.

Customizing the CheckBox component

You can transform a CheckBox component horizontally and vertically both 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 (see 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 clips 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 non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

A CheckBox component supports the following Halo styles:

Style	Description
`themeColor`	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.

Style	Description
fontSize	The point size for the font.
fontStyle	The font style: either "normal", or "italic".
fontWeight	The font weight: either "normal", or "bold".
textDecoration	The text decoration: either "none", or "underline".

Using skins with the CheckBox component

The CheckBox component uses symbols in the Library panel 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 36.

A CheckBox component uses the following skin properties:

Property	Description
falseUpSkin	The up state. Default is RectBorder.
falseDownSkin	The pressed state. Default is RectBorder.
falseOverSkin	The over state. Default is RectBorder.
falseDisabledSkin	The disabled state. Default is RectBorder.
trueUpSkin	The toggled state. Default is RectBorder.
trueDownSkin	The pressed-toggled state. Default is RectBorder.
trueOverSkin	The over-toggled state. Default is RectBorder.
trueDisabledSkin	The disabled-toggled state. Default is RectBorder.

CheckBox class

Inheritance UIObject > UIComponent > SimpleButton > Button > CheckBox

ActionScript Class Name mx.controls.CheckBox

The properties of the CheckBox class allow you to 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 panel.

The CheckBox component uses the FocusManager 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 24.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.CheckBox.version);
```

Note: The following code returns undefined: `trace(myCheckBoxInstance.version);`.

Method summary for the CheckBox class

Inherits all methods from UIObject and UIComponent.

Property summary for 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`).

Inherits all properties from UIObject and UIComponent.

Event summary for the CheckBox class

Event	Description
`CheckBox.click`	Triggered when the mouse is pressed over a button instance.

Inherits all events from UIObject and UIComponent.

CheckBox.click

Availability

Flash Player 6 version 79.

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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `addEventListener()` method (see `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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` action 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 the `addEventListener()` method from `checkBoxInstance` and passes it the `click` event and the `form` listener object as parameters, as in the following:

```
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`, as in the following:

```
on(click){
  trace("check box component was clicked");
}
```

See also

`UIEventDispatcher.addEventListener()`

CheckBox.label

Availability

Flash Player 6 version 79.

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 clip parameters 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 version 79.

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 top 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 grouping are centered horizontally and vertically.

- `"top"` The label is placed below the check box. The check box and label grouping are centered horizontally and vertically.

You can change the bounding area of 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 85.

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 version 79.

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;
```

ComboBox component

A combo box can be static or editable. A static combo box allows a user to make a single selection from a drop-down list. 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 516), with the exception of the following keys:

Key	Description
Control+Down	Opens the drop-down list and gives it focus.
Shift +Tab	Moves focus to the previous object.
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	Opens the drop-down list and gives it focus.
Control+Up	Closes the drop-down list, if open in the Stand alone and Browser versions of the Flash Player.
Down	Selection moves 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	Selection moves to the top of the list.
Page Down	Selection moves down one page.
Page Up	Selection moves 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	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 the Flash Player.
Down	Selection moves down one item.
End	The insertion point moves 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	The insertion point moves to the beginning of the text box.
Page Down	Selection moves down one page.
Page Up	Selection moves 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	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 24 or "FocusManager class" on page 270.

A live preview of each ComboBox component instance on the Stage reflects changes made to parameters in the Property inspector or Component Inspector panel while 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 only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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 a driving directions application 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

The following are authoring parameters that you can set for each ComboBox component instance in the Property inspector or in the Component Inspector panel:

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 at one time without using a scroll bar. 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.

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, do the following:

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 while 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 panel or the 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. 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:

```
form = new Object();
form.change = function (evt){
  trace(evt.target.selectedItem.label);
}
comboBox.addEventListener("change", form);
```

   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 clips 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.

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 non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

The combo box has two unique styles. 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 68.)
- The text is a TextInput instance and uses its styles. (See "Using styles with the TextInput component" on page 518.)
- The drop-down list is an List instance and uses its styles. (See "Using styles with the List component" on page 289.)

A ComboBox component uses the following Halo styles:

Style	Description
`themeColor`	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style: either "normal", or "italic".
`fontWeight`	The font weight: either "normal", or "bold".
`textDecoration`	The text decoration: either "none", or "underline".
`openDuration`	The number of milliseconds to open the drop-down list. The default value is 250.
`openEasing`	A reference to a tweening function that controls the drop-down list animation. Defaults to sine in/out. For more equations, download a list from Robert Penner's website at www.robertpenner.com/easing/.

Using skins with the ComboBox component

The ComboBox component uses symbols in the Library panel to represent the button states. The ComboBox has skin variables for the down arrow. Other than that, it uses scroll bar and list skins. To skin the ComboBox component while authoring, modify symbols in the Library panel and re-export the component as a SWC. The CheckBox component skins are located in the Flash UI Components 2/Themes/MMDefault/ComboBox 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 36.

A ComboBox component uses the following skin properties:

Property	Description
`ComboDownArrowDisabledName`	The down arrow's disabled state. Default is `RectBorder`.
`ComboDownArrowDownName`	The down arrow's down state. Default is `RectBorder`.
`ComboDownArrowUpName`	The down arrow's up state. Default is `RectBorder`.
`ComboDownArrowOverName`	The down arrow's over state. Default is `RectBorder`.

ComboBox class

Inheritance UIObject > UIComponent > ComboBase > ComboBox

ActionScript Class Name mx.controls.ComboBox

The ComboBox component combines three separate subcomponents: Button, TextInput, and List. Most of the APIs of each subcomponent are available directly from ComboBox component and are listed in the Method, Property, and Event tables for the ComboBox class.

The drop-down list in a combo box is provided either as an Array or as a DataProvider object. If you use a DataProvider object, the list changes at runtime. The source of the ComboBox data can be changed dynamically by switching to a new Array or DataProvider object.

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 component 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.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.ComboBox.version);
```

Note: The following code returns undefined: `trace(myComboBoxInstance.version);`.

Method summary for the ComboBox class

Property	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.
ComboBox.removeItemAt()	Removes an item from the list at the specified location.
ComboBox.replaceItemAt()	Replaces an item in the list with another specified item.

Inherits all methods from UIObject and UIComponent.

Property summary for 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 or not 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.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 the 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).

Inherits all properties from UIObject and UIComponent.

Event summary for 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 drop-down list begins to close.
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.

Inherits all events from UIObject and UIComponent.

ComboBox.addItem()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

Usage 1:

```
comboBoxInstance.addItem(label[, data])
```

Usage 2:

```
comboBoxInstance.addItem({label:label[, data:data]})
```

Usage 3:

```
comboBoxInstance.addItem(obj);
```

Parameters

label A string that indicates the label for the new item.

data The data for the item; 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 version 79.

Edition

Flash MX 2004.

Usage

```
comboBoxInstance.addItemAt(index, label[, data])
```

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; can be any data type. This parameter is optional.

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 version 79.

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 value of the combo box changes as a result of user interaction.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox 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 ComboBox component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `addEventListener()` method (see `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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

`UIEventDispatcher.addEventListener()`

ComboBox.close()

Availability

Flash Player 6 version 79.

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 version 79.

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 list of the combo box begins to retract.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox 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 ComboBox component 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 a set of properties that contains 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 about event objects, see "Event Objects" on page 562.

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

`UIEventDispatcher.addEventListener()`

ComboBox.dataProvider

Availability

Flash Player 6 version 79.

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 interface. The default value is []. This is a property of the List component but can be accessed directly from an instance of the ComboBox component.

The List component, and other data-aware components, add methods to the Array object's prototype so that they conform to the DataProvider interface (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` properties are 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 lose the selection.

Any instance that implements the DataProvider interface is eligible as a data provider for a List. This includes Flash Remoting RecordSets, Firefly DataSets, 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, as in the following:

```
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 version 79.

Edition

Flash MX 2004.

Usage

myComboBox.dropdown

Description

Property (read-only); returns a reference to the List component 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 version 79.

Edition

Flash MX 2004.

Usage

myComboBox.change

Description

Property; the width limit in pixels of the drop-down list. The default value is the width of the ComboBox component (the TextInput instance plus the SimpleButton instance).

Example

The following code sets the `dropdownWidth` to 150 pixels:

```
myComboBox.dropdownWidth = 150;
```

See also

`ComboBox.dropdown`

ComboBox.editable

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

myComboBox.editable

Description

Property; indicates whether the combo box is editable (`true`) or not (`false`). An editable combo box can have values entered into the text box that do not show up in the drop-down list. If a combo box is not editable, only values listed in the drop-down list can be entered into the text box. The default value is `false`.

Setting a combo box to 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 version 79.

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 Enter key has been pressed in the text box. This event is only broadcast from editable combo boxes. This is a TextInput event that is broadcast from a combo box. For more information, see `TextInput.enter`.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox 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 ComboBox component 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 a set of properties that contains 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 about event objects, see "Event Objects" on page 562.

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

`UIEventDispatcher.addEventListener()`

ComboBox.getItemAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`comboBoxInstance``.getItemAt(`*`index`*`)`

Parameters

`index` A number greater than or equal to 0, and less than `ComboBox.length`. The index of the item to retrieve.

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 version 79.

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 additional property: `index`. The `index` is the number of the item that was rolled out.

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 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 ComboBox component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

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 which item index number has been rolled off of:

```
form.itemRollOut = function (eventObj) {
  trace("Item #" + eventObj.index + " has been rolled out of.");
}
myCombo.addEventListener("itemRollOut", form);
```

See also

ComboBox.itemRollOver, UIEventDispatcher.addEventListener()

ComboBox.itemRollOver

Availability

Flash Player 6 version 79.

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 additional property: `index`. The `index` is the number of the item that was rolled over.

Description

Event; broadcast to all registered listeners when the drop-down list items are rolled over. 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 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 ComboBox component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

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 which item index number has been rolled over:

```
form.itemRollOver = function (eventObj) {
  trace("Item #" + eventObj.index + " has been rolled over.");
}
myCombo.addEventListener("itemRollOver", form);
```

See also

`ComboBox.itemRollOut`, `UIEventDispatcher.addEventListener()`

ComboBox.labelField

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

myComboBox.labelFunction

Description

Property; a function that computes the label of a dataProvider 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 version 79.

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 an instance of ComboBox. 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 version 79.

Edition

Flash MX 2004.

Usage

```
myComboBox.open()
```

Parameters

None.

Returns

Nothing.

Description

Property; 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 version 79.

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 begins to appear.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox 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 ComboBox component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

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 which item index number has been rolled out:

```
form.open = function () {
  trace("The combo box has opened with text " + myBox.text);
}
myBox.addEventListener("open", form);
```

See also

`ComboBox.close`, `UIEventDispatcher.addEventListener()`

ComboBox.removeAll()

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

listInstance.removeItemAt(*index*)

Parameters

index A number that indicates the position of the item to remove. This value 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 version 79.

Edition

Flash MX 2004.

Usage

```
comboBoxInstance.replaceItemAt(index, label[, data])
```

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.

Returns

Nothing.

Description

Method; replaces the content of the item at the index specified by the *index* parameter. 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.rowCount

Availability

Flash Player 6 version 79.

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 or equal to the `rowCount` property, it 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 version 79.

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.

The first usage example uses an `on()` handler and must be attached directly to a ComboBox 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

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 which item index number has been scrolled to:

```
form.scroll = function (eventObj) {
  trace("The list had been scrolled to item # " + eventObj.target.vPosition);
}
myCombo.addEventListener("scroll", form);
```

See also

`UIEventDispatcher.addEventListener()`

ComboBox.selectedIndex

Availability

Flash Player 6 version 79.

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 that label of the indicated item in the combo box's text box.

Assigning a `selectedIndex` that is out of range is ignored. Entering text into the text field of an editable combo box sets `selectedIndex` to undefined.

Example

The following selects the last item in the list:

```
myComboBox.selectedIndex = myComboBox.length-1;
```

See also

`ComboBox.selectedItem`

ComboBox.selectedItem

Availability

Flash Player 6 version 79.

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 you enter any text in the text box. It will only have a value if you select an item from the drop-down list, or the value is set via ActionScript. If the combo box is static, the value of `selectedItem` is always valid.

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.text

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

myComboBox.textField

Description

Property (read-only); a reference to the TextInput component contained by the ComboBox.

This property allows you to access the underlying TextInput component so that you can to manipulate it. For example, you might want to change the selection of the text box or restrict the characters that can be entered into it.

Example

The following code restricts the text box of `myComboBox` to only accept numbers:

```
myComboBox.textField.restrict = "0-9";
```

ComboBox.value

Availability

Flash Player 6 version 79.

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, as follows:

```
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 using the Bindings tab in the Component Inspector panel, or you can programmatically create and configure bindings 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 Help.

Making data binding classes available at runtime (Flash Professional only)

In order to make the data binding service classes available at runtime, the DataBindingClasses component must be in your FLA file's library. When you visually create bindings in the Flash authoring environment, this component is automatically added to your document's library. But if you're only using ActionScript to create bindings at runtime, then you have to add this component manually to your document's library. For information on how to add this component to your document, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

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-specific functionality to components.
CustomFormatter class (Flash Professional only)	Base class for creating custom formatter classes.
CustomValidator class (Flash Professional only)	Base class for creating custom validator classes.
DataType class (Flash Professional only)	Provides read and write access to data fields of a component property.
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 using the Binding class (and supporting classes), or use the Bindings tab in the Component Inspector panel (Window > Development Panels > Component Inspector).

Note: To make this class available at runtime, you must include the DataBindingClasses component in your FLA document. For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

For an overview of the classes in the mx.data.binding package, see "Data binding classes (Flash Professional only)" on page 118.

Method summary for 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 version 79.

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)).

`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 (see EndPoint class (Flash Professional only)).

`format` (Optional) An 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` (Optional) A 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 inner-most 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, although you can. 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

Example #1: 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);
```

Example #2: This example demonstrates how to create a Binding object that uses a custom formatter class. For more information on creating custom formatter classes, see "CustomFormatter class (Flash Professional only)" on page 121.

```
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, returns an array of error messages (strings) that describe the error, or 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 118), or by using the Bindings tab in the Component Inspector panel. For information on assigning a formatter class using the Component Inspector, see "Schema formatters (Flash Professional only)" in Using Flash Help.

You can also assign a formatter class to a component property on the Component Inspector panel's Schema tab. However, in that case, the formatter will only get used when the data is needed in the form of a string. In contrast, formatters assigned using 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 122.

Note: To make this class available at runtime, you must include the DataBindingClasses component in your FLA document. For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

For an overview of the classes in the mx.data.binding package, see "Data binding classes (Flash Professional only)" on page 118.

Sample custom formatter

The following example demonstrates how to create a custom formatter class and then apply it to a binding between two components 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, returns 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 Open the Components panel (Window > Development Panels > Components).

6 Drag a TextInput component to the Stage and name it **textInput**.

7 Drag a NumericStepper component to the Stage and name it **stepper**.

8 Open the Timeline (Window > Timeline) and select the first frame on Layer 1.

9 Open the Actions panel (Window > Development Panels > Actions).

10 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.

11 Select Window > Panels > Other Panels > Classes to open the Classes library.

12 Open your document's library by selecting Window > Library.

13 Drag the DataBindingClasses component from the Classes library to your document's library.

This makes the data binding runtime classes available for your document. .For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

14 Save the FLA file to the same folder that contains NumberFormatter.as.

15 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

Method	Description
`CustomFormatter.format()`	Converts from a raw datatype to a text string.
`CustomFormatter.unformat()`	Converts from a text string to a raw datatype.

CustomFormatter.format()

Availability

Flash Player 6 version 79.

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 this method in your subclass of mx.data.binding.CustomFormatter.

Example

See "Sample custom formatter" on page 122.

CustomFormatter.unformat()

Availability

Flash Player 6 version 79.

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 perform the exact inverse transformation of the `CustomFormatter.format()`.

This method is not implemented by default. You must define this method in your subclass of mx.data.binding.CustomFormatter.

For more information, see "Sample custom formatter" on page 122.

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 validation 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 using the Component Inspector panel's Schema tab. For an example of creating and using a custom validator class, see the Example section in the entry for `CustomValidator.validate()`.

To assign a custom validator, do the following:

1. In the Component Inspector panel (Window > 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 validation class you created.

   In order for the class you specify to be included in the published SWF, it must be in the classpath.

Note: To make this class available at runtime, you must include the DataBindingClasses component in your FLA document. For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

For an overview of the classes in the mx.data.binding package, see "Data binding classes (Flash Professional only)" on page 118.

Method summary for the CustomValidator class

Method	Description
`CustomValidator.validate()`	Performs validation on data.
`CustomValidator.validationError()`	Reports validation errors.

CustomValidator.validate()

Availability

Flash Player 6 version 79.

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 you like 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 the `validate()` method might be called repeatedly, you should avoid adding code to this method 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 example below).

Example

The following procedure demonstrates how to create and use a custom validation class. The `validate()` method of the CustomValidator class, OddNumbersOnly.as, determines as invalid any value that not an odd number. The validation occurs whenever the value of a NumericStepper component changes, 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 a 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 a 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 (Window > Development Panels > Components).

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 panel (Window > Development Panels > Component Inspector).

10 Select the Bindings tab in the Component Inspector panel and click the Add Binding (+) button.

11 Select the `Value` property (the only one) in the Add Bindings dialog, then click OK

12 In the Component Inspector panel, 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 the 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 (Window > Timeline) and select the first frame on Layer 1.

19 Open the Actions panel (Window > Actions).

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 (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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
this.validationError(errorMessage)
```

Note: This method can be invoked only from inside 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; you call this method from the `validate()` method of your subclass of CustomValidator to report validation errors. If you don't call this method, then a `valid` event is generated when `validate()` completes. If you call this method one or more times from within the `validate()` method then 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 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 a 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, namely `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 DataBindingClasses component in your FLA document. For more information, see “Working with data binding and web services at runtime (Flash Professional only)” in Using Flash Help.

For an overview of the classes in the mx.data.binding package, see "Data binding classes (Flash Professional only)" on page 118.

Property summary for the EndPoint class

Method	Description
`EndPoint.constant`	A constant value.
`EndPoint.component`	A reference to a component instance.
`EndPoint.property`	The name of a property of the component instance specified by `EndPoint.component`.
`EndPoint.location`	The location of a data field within the property of the component instance.
`EndPoint.event`	The name of an event, or list of events, the component instance will emit when the data changes.

Constructor for the EndPoint class

Availability

Flash Player 6 version 79.

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_txt` 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.constant

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

endPoint_src.constant

Description

Property; a constant value assigned to an EndPoint object. This property can only be applied to EndPoints that are the source, not the destination, of a binding between components. The value can be any data type that is compatible with the destination of the binding. If 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.component

Availability

Flash Player 6 version 79.

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 a EndPoint object.

```
var sourceEndPoint = new mx.data.binding.EndPoint();
sourceEndPoint.component=listBox1;
```

EndPoint.property

Availability

Flash Player 6 version 79

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);
```

EndPoint.location

Availability

Flash Player 6 version 79.

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 either an XPath expression or an ActionScript path, an array of strings, or an object.

XPath expressions can only be used when the data is an XML object. For a list of supported XPath expressions, see "Supported XPath expressions" in Using Flash Help. (See Example 1 below.)

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 using an ActionScript; 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 being evaluated, the indices are used to index into arrays. The index item can be any EndPoint. 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 string to "drill down" to a nested movie clip property.

```
var sourceEndPoint = new mx.data.binding.EndPoint();
//assume movieClip1.ball.position exists
ssourceEndPoint.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.event

Availability

Flash Player 6 version 79.

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 118.

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` events are 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);
```

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 DataBindingClasses component in your FLA document. For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

For an overview of the classes in the mx.data.binding package, see "Data binding classes (Flash Professional only)" on page 118.

Method summary for 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 ComponentMixin methods to a component.
`ComponentMixins.refreshFromSources()`	Executes all bindings that have this component as the destination EndPoint.
`ComponentMixins.refreshDestinations()`	Executes all the bindings that have this object as the source EndPoint.
`ComponentMixins.validateProperty()`	Checks to see if the data in the indicated property is valid.

ComponentMixins.getField()

Availability

Flash Player 6 version 79.

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 (Optional) The location of a field within the component property. This is useful if the component property specified by propertyName is a complex data structure and you are interested in a particular field of that structure. This property can take one of the following three forms:

- A string that contains a XPath expression. This is only valid for XML data structures. For a list of supported XPath expressions, see "Supported XPath expressions" in Using Flash Help.
- 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 138.

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 version 79.

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 not involved in a data binding, you must explicitly call this method for that component.

Example

The following code makes the ComponentMixins methods avaialble to a DataSet component.

```
mx.data.binding.ComponentMixins.initComponent(_root.myDataSet);
```

ComponentMixins.refreshFromSources()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

componentInstance.refreshSources()

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 any "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.refreshDestinations()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

componentInstance.refreshDestinations()

Returns

Nothing.

Description

Method; executes all the bindings for which *componentInstance* is the source EndPoint. 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.validateProperty()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

componentInstance.validateProperty(*propertyName*)

Parameters

propertyName A string that contains the name of a property belonging 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, returns an array of error messages as strings.

Validation only applies to fields that have schema information available. If a field is an object that contains other fields, then each "child" field will be validated, and so on, recursively. Each individual field will dispatch a `valid` or `invalid` event, as necessary. For each data field contained by *propertyName*, this function 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 non-null and the field's schema does *not* have a validator, the method returns `null`; no events are generated.
- If the value is non-null and the field's schema *does* define a validator, then 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 examples shows how to use `validateProperty()` to make sure that text entered by a user is of a valid length. You'll determine what a valid length is by setting the Validation Options for the String DataType in the Component Inspector panel's Schema tab. If the user enters a string in the text field of an invalid length, the error messages returned by the `validateProperty()` method 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 (Window > Development Panels > Components) to the Stage, and name it **zipCode_txt**.
2. Select the TextInput component and, in the Component Inspector panel (Window > Development Panels > Components), 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 not already open, by choosing Window > Timeline.
6. Click the first frame on Layer 1 in the Timeline, and open the Actions panel (Window > 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 the DataBindingClasses component from the Classes library to your document's Library panel.

    This step is required to make the data binding runtime classes available to the SWF at runtime. For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.
11. Test the SWF by choosing 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 less 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()` function on a component. You can then call methods of the DataType object to get and set the value of the field.

The difference between getting and setting field values using DataType object methods, and getting or setting the same values directly on the component instance, is that the latter case provides the data in its "raw" form. In contrast, when you get or set field values using methods of the DataType class, those 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 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 Help.

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 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 using the `ComponentMixins.initComponent()` method. For more information, see "ComponentMixins class (Flash Professional only)" on page 133.

Note: To make this class available at runtime, you must include the DataBindingClasses component in your FLA document. For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

For an overview of the classes in the mx.data.binding package, see "Data binding classes (Flash Professional only)" on page 118.

Method summary for the DataType class

Method	Description
`DataType.getAsBoolean()`	Fetches the current value of the field as a Boolean.
`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.getAnyTypedValue()`	Fetches the current value of the field.

Method	Description
DataType.getTypedValue()	Fetches the current value of the field in the form of the requested DataType.
DataType.setAnyTypedValue()	Sets a new value into the field.
DataType.setAsBoolean()	Sets the field to the new value, which is given as a Boolean.
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 into the field.

Property summary for the DataType class

Property	Description
DataType.encoder	Provide 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 version 79.

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 Schema tab of the Component Inspector panel.

If no encoder was applied to the field in question, then this property will return `undefined`.

For more information about the encoders provided with Flash MX Professional 2004, see "Schema encoders (Flash Professional only)" in Using Flash Help.

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` Boolean values. The code below sets the `trueStrings` property for a field's encoder to be the strings "yes" and "si".

```
var myField:mx.data.binding.DataType = dataSet.getField("isValid");
myField.encoder.trueStrings = "Yes,Oui";
```

DataType.formatter

Availability

Flash Player 6 version 79.

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 Schema tab of the Component Inspector panel.

If no formatter was applied to the field in question, then this property will return `undefined`.

For more information about the encoders provided with Flash MX Professional 2004, see "Schema formatters (Flash Professional only)" in Using Flash Help.

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.getAsBoolean()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.getAsBoolean()

Returns

A Boolean value.

Description

Method; fetches the current value of the field as a Boolean. The value is converted 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()`

Returns

A number.

Description

Method; fetches the current value of the field as a number. The value is converted 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 version 79.

Edition

Flash MX Professional 2004.

Usage

`dataTypeObject``.getAsString()`

Returns

A string.

Description

Method; fetches the current value of the field as a string. The value is converted to String form, if necessary.

Example

In this example, a property of a component 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.getAnyTypedValue()

Availability

Flash Player 6 version 79.

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 you'd like for the field. For more information, see the Description section below.

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 is able to provide a value as the first data type specified in the *`suggestedTypes`* array, then 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, then the method returns the value of the field in the data type specified in the Schema panel. Specifying `null` will always result 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, then it is returned in the type specified in the Schema panel.

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.getTypedValue()

Availability

Flash Player 6 version 79.

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 147)

Description

Method; returns the value of the field in the form specified by *`requestedType`*, if specified and if the field can provide its value in that form. If the field isn't able to provide its value in the requested form then the method returns `null`.

If `null` is specified as the *`requestedType`* then the method returns the value of the field in its default type.

Example

```
var bool:TypedValue = field.getTypedValue("Boolean");
```

DataType.kind

Availability

Flash Player 6 version 79.

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 to access properties and methods of the Kind object.

DataType.setAnyTypedValue()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.setAnyTypedValue(*newTypedValue*)

Parameters

newValue A TypedValue object value to set into the field.

For more information about TypedValue objects, see "TypedValue class (Flash Professional only)" on page 147.

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 the message(s) will vary depending on the data type, formatters, and encoders that are defined in the field's schema.

Description

Method; sets a new value into 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 version 79.

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. The value is converted to, and stored as, the data type that is appropriate for this field.

Example

```
var bool: Boolean = true;
field.setAsBoolean (bool);
```

DataType.setAsNumber()

Availability

Flash Player 6 version 79.

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

```
var num: Number = 32;
field.setAsNumber (num);
```

DataType.setAsString()

Availability

Flash Player 6 version 79.

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

```
var stringVal: String = "The new value";
field.setAsString (stringVal);
```

DataType.setTypedValue()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

dataTypeObject.setTypedValue(*newTypedValue*)

Parameters

newValue A TypedValue object value to set into the field.

For more information about TypedValue objects, see "TypedValue class (Flash Professional only)" on page 147.

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 datatype 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 the message(s) will vary depending on the data type, formatters, and encoders that are defined in the field's schema.

Description

Method; sets a new value into 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 is an object that 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 DataBindingClasses component in your FLA document. For more information, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

For an overview of the classes in the mx.data.binding package, see "Data binding classes (Flash Professional only)" on page 118.

Property summary for the TypedValue class

Property	Description
`TypedValue.type`	Contains the schema associated with the TypedValue object's value.
`TypedValue.typeName`	Contains the name of the DataType 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 version 79.

Usage

```
new mx.data.binding.TypedValue(value, typeName, [type])
```

Parameters

value A data value. This can be any type.

typeName A String that contains the name of the DataType of the value.

type (Optional) A Schema object that describes in more detail the schema of the data. This field is only required 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
typedValueObject.type
```

Description

Property; contains the schema associated with the TypedValue object's value. It is only used in certain circumstances.

Example

This example will display "null" in the Output panel.

```
var t: TypedValue = new TypedValue (true, "Boolean", null);
trace(t.type);
```

TypedValue.typeName

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
typedValueObject.typeName
```

Description

Property; contains the name of the DataType of the TypedValue object's value.

Example

This example will display "Boolean" in the Output panel.

```
var t: TypedValue = new TypedValue (true, "Boolean", null);
trace(t.typeName);
```

TypedValue.value

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

typedValueObject.value

Description

Property; contains the data value of the TypedValue object.

Example

This example will display "true" in the Output panel.

```
var t: TypedValue = new TypedValue (true, "Boolean", null);
trace(t.value);
```

DataGrid component (Flash Professional only)

The DataGrid component allows you to create powerful data-enabled displays and applications. You can use the DataGrid component to instantiate a recordset (retrieved from a database query in 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 v2 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, no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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` is `true` and `DataGridColumn.sortOnHeaderRelease` is `true`, clicking within 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 allows you to resize columns.

Clicking within 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 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 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 does nothing.
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 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 wraps from the last column in the grid to the first column on the next line. When Shift+Tab is pressed, wrapping is reversed.

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

The DataGrid component consists of two sets of APIs: the DataGrid class and the DataGridColumn class.

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 allow you to manipulate data and broadcast changes to multiple views. Any object that implements the DataProvider interface 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 contents 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 within 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

  This consists of the view objects (instances 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 interface (see "CellRenderer API" on page 77). For a basic data grid, a cell is a built-in ActionScript TextField object.

DataGrid parameters

The following are authoring parameters that you can set for each DataGrid component instance in the Property inspector or in the Component Inspector panel:

multipleSelection A Boolean value that indicates whether multiple items can be selected (`true`) or not (`false`). The default value is `false`.

rowHeight The height of each row, in pixels. Changing the font size does not change the row height. The default value is 20.

editable 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 154.

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 class.

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, then 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 Halo styles from the List component. (For more information, see "Using styles with the List component" on page 289.) The DataGrid component also supports the following Halo styles:

Style	Description
`backgroundColor`	The background color can be set for the whole grid or for each column.
`labelStyle`	The font style can be set for the whole grid or for each column.
`headerStyle`	A CSS Style Declaration for the column header that can be applied to a grid or column.
`vGridLines`	A Boolean value that indicates whether to show vertical grid lines (`true`) or not (`false`).
`hGridLines`	A Boolean value that indicates whether to show horizontal grid lines (`true`) or not (`false`).
`vGridLineColor`	The color of the vertical grid lines.
`hGridLineColor`	The color of the horizontal grid lines.
`headerColor`	The color of the column headers.

If the above table indicates that a style can be set for a column, you can use the following syntax to set the style:

```
grid.getColumnAt(3).setStyle("backgroundColor", 0xff00aa)
```

Using skins with the DataGrid component

The skins that the DataGrid component uses to represent its visual states are included in the subcomponents from which the data grid is composed (ScrollPane and RectBorder). For information about their skins, see "Using skins with the ScrollPane component" on page 466 and "Using skins with the List component" on page 290.

The rollover and selection underlays, however, use the ActionScript Drawing API. To skin these portions of the data grid while authoring, modify the ActionScript code in the skin symbols in the Flash UI Components 2/Themes/MMDefault/datagrid/ skins states folder in the library of one of the themes FLA files. For more information, see "About skinning components" on page 36.

DataGrid class (Flash Professional only)

Inheritance mx.core.UIObject > mx.core.UIComponent > mx.core.View > mx.core.ScrollView > mx.controls.listclasses.ScrollSelectList > mx.controls.List

ActionScript Class Name mx.controls.DataGrid

Each component class has a `version` property, which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.DataGrid.version);
```

Note: The following code returns `undefined`: `trace(myDataGridInstance.version);`.

Method summary for 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 specific location.
`DataGrid.addItem()`	Adds an item to the data grid.
`DataGrid.addItemAt()`	Adds an item to the data grid at a specific 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 the index of the column.
`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.

Inherits all properties from UIObject and UIComponent.

Property summary for 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`).

Event summary for 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.
DataGrid.change	Broadcast when an item has been selected.
DataGrid.columnStretch	Broadcast when a column is resized by a user.
DataGrid.headerRelease	Broadcast when a user presses and releases a header.

DataGrid.addColumn()

Availability

Flash Player 6 version 79.

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 174.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myDataGrid.addColumnAt(index, name)
```

Usage 2:

```
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. You must specify either the *index* parameter or the *dataGridColumn* parameter.

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 174.

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 version 79.

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};
var addedObject = myGrid.addItem(anObject);
```

DataGrid.addItemAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.addItemAt(index, item)
```

Parameters

index The order (among the child nodes) in 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};
var addedObject = myGrid.addItemAt(4, anObject);
```

DataGrid.cellEdit

Availability

Flash Player 6 version 79.

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 has changed.

V2 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 a set of 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 "Event Objects" on page 562.

Example

In the following example, a handler called `myDataGridListener` is defined and passed to the `myDataGrid.addEventListener()` method 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, as follows:

```
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 version 79.

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.

V2 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 a set of 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 "Event Objects" on page 562.

Example

In the following example, a handler called `myListener` is defined and passed to the `grid.addEventListener()` method 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, as follows:

```
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 version 79.

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.

V2 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 a set of 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 "Event Objects" on page 562.

Example

In the following example, a handler called `myListener` is defined and passed to the `grid.addEventListener()` method 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, as follows:

```
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 version 79.

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.

V2 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 a set of 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 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 `"cellPress"`.

For more information, see "Event Objects" on page 562.

Example

In the following example, a handler called `myListener` is defined and passed to the `grid.addEventListener()` method 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, as follows:

```
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 version 79.

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.

V2 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 a set of 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`, and its value is `"change"`. For more information, see "Event Objects" on page 562.

Example

In the following example, a handler called `myListener` is defined and passed to the `grid.addEventListener()` method 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, as follows:

```
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 version 79.

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 version 79.

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 version 79.

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 horizontally resizes a column.

V2 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 a set of 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 "Event Objects" on page 562.

Example

In the following example, a handler called `myListener` is defined and passed to the `grid.addEventListener()` method 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, as follows:

```
var myListener = new Object();
myListener.columnStretch = function(event) {
    trace("column " + event.columnIndex + " was resized");
};
grid.addEventListener("columnStretch", myListener);
```

DataGrid.dataProvider

Availability

Flash Player 6 version 79.

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 interface (see DataProvider.as in the Classes/mx/controls/listclasses folder). Any array that exists 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 the `DataGrid.addColumn()` method) 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 interface can be used as a data provider for a data grid (including Flash Remoting recordsets, data sets, and arrays).

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 class. 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 version 79.

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`.

Example

The following example sets the scroll position to the top of the display:

```
myDataGrid.editable = true;
```

DataGrid.editField()

Availability

Flash Player 6 version 79.

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.

Example

The following example places a value in the grid:

```
var prevValue = myGrid.editField(5, "Name", "Neo");
```

DataGrid.focusedCell

Availability

Flash Player 6 version 79.

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 version 79.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.getColumnIndex(index)
```

Parameters

index The index of the DataGridColumn object to be returned.

Returns

A DataGridColumn object.

Description

Method; gets a reference to the DataGridColumn object at the specified index.

DataGrid.headerHeight

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.headerHeight
```

Description

Property; the height of the header bar of the data grid. 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 version 79.

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 allow you to sort as you like.

V2 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 a set of 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 "Event Objects" on page 562.

Example

In the following example, a handler called `myListener` is defined and passed to the `grid.addEventListener()` method 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, as follows:

```
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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDataGrid.hScrollPolicy
```

Description

Property; specifies whether the data grid has a horizontal scroll bar. This property can have one of three values: `"on"`, `"off"`, and `"auto"`. The default value is `"off"`.

If you set `hScrollPolicy` to `"off"`, columns scale proportionally to accommodate the finite width.

Example

The following example sets horizontal scroll policy to automatic:

```
myDataGrid.hScrollPolicy = "auto";
```

DataGrid.removeAllColumns()

Availability

Flash Player 6 version 79.

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.

Example

The following example removes all DataGridColumn objects from `myDataGrid`:

```
myDataGrid.removeAllColumns();
```

DataGrid.removeColumnAt()

Availability

Flash Player 6 version 79.

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 version 79.

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.

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 version 79.

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 viewer (`true`) or not (`false`). This property must be `true` for individual columns to be resizable. The default value is `true`.

Example

The following example prevents users from resizing columns:

```
myDataGrid.resizableColumns = false;
```

DataGrid.selectable

Availability

Flash Player 6 version 79.

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 version 79.

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 is `true`.

Example

The following example hides the column headers:

```
myDataGrid.showHeaders = false;
```

See also

`DataGrid.sortableColumns`

DataGrid.sortableColumns

Availability

Flash Player 6 version 79.

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. This property must be set to `true` in order to broadcast the `headerRelease` event. The default value is `true`.

Example

The following example turns off sorting:

```
myDataGrid.sortableColumns = false;
```

See also

`DataGrid.headerRelease`

DataGrid.spaceColumnsEqually()

Availability

Flash Player 6 version 79.

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.gridclassesDataGridColumn

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 call `DataGrid.getColumnAt(`*`index`*`)` to access them.

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 DataProvider 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 allows you to 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 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 175.

Property summary for 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 presses a column header.
`DataGridColumn.width`	The width of a column, in pixels.

Constructor for the DataGridColumn class

Availability

Flash Player 6 version 79.

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 version 79.

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 interface (see "CellRenderer API" on page 77.) 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 version 79.

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 version 79.

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`.

Example

The following example makes the first column in a grid uneditable:

```
myDataGrid.getColumnAt(0).editable = false;
```

See also

`DataGrid.editable`

DataGridColumn.headerRenderer

Availability

Flash Player 6 version 79.

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 interface (see "CellRenderer API" on page 77). 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 version 79.

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.

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 version 79.

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.

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 version 79.

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 version 79.

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`.

Example

The following example prevents the column at index 1 from being sorted:

```
myGrid.getColumnAt(1).sortable = false;
```

DataGridColumn.sortOnHeaderRelease

Availability

Flash Player 6 version 79.

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 on 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`.

Example

The following example allows you to catch the `headerRelease` event to perform your own sort:

```
myGrid.getColumnAt(7).sortOnHeaderRelease = false;
```

DataGridColumn.width

Availability

Flash Player 6 version 79.

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 size of the default value:

```
myGrid.getColumnAt(4).width = 25;
```

DataHolder component (Flash Professional only)

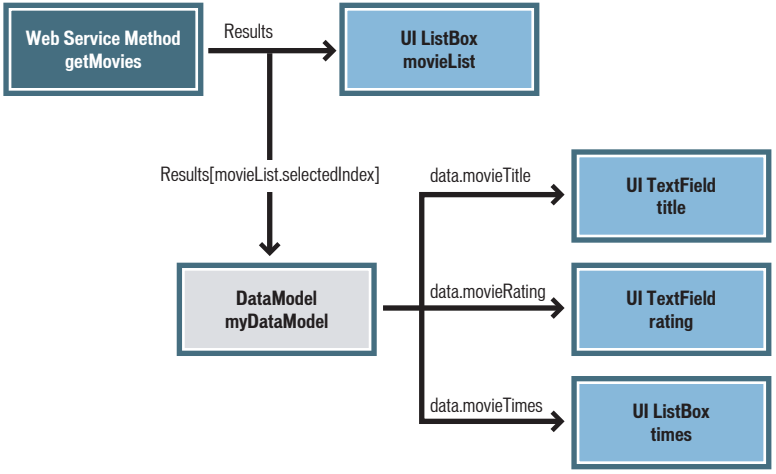
The DataHolder component is a repository for data and a means of generating events when that data has changed. Its main purpose is to hold data and act as connector between other components using data binding.

Initially, the DataHolder component has a single bindable property named `data`. You can add more properties using the Schema tab in the Component Inspector panel (Window > Development Panels > Component Inspector). For more information on using the Schema tab, see "Working with schemas in the Schema tab (Flash Professional only)" in Using Flash Help.

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.

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 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.

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 panel.

To use the DataHolder component in a simple application:

1. In Flash MX Professional 2004, create a new file.
2. Open the Components panel (Window > Development Panels > Components), 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 panel (Window > Development Panels > Component Inspector).
5. Click the Schema tab in the Component Inspector panel.
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 panel, 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 Help.
9. In the Timeline (Window > Timeline), select the first frame on Layer 1 and open the Actions panel (Window > Development Panels > Actions).
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.

Property summary for the DataHolder class

Property	Description
`DataHolder.data`	Default bindable property for DataHolder component.

DataHolder.data

Availability

Flash Player 6 version 79.

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 panel.

For more information on using the Schema tab, see "Working with schemas in the Schema tab (Flash Professional only)" in Using Flash Help.

Example

For an example of using this component, see "Creating an application with the DataHolder component (Flash Professional only)" on page 182.

DataProvider API

ActionScript class name mx.controls.listclasses.DataProvider

The DataProvider API is a set of methods and properties that a data source needs to have in order to have a List-based class communicate with it. Arrays, RecordSets, and the DataSet all implement this API. You can create a DataProvider-compliant class by implementing all the methods and properties described in this document. A List-based component could then use that class as a data provider.

The methods of the DataProvider API allow you to 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 all 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 through their index (as you can with an array), using `DataProvider.getItemAt()`.

The most common case for using data providers is 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 allows you to use any existing array as the data for views that have a `dataProvider` property.

Because of the DataProvider API, the v2 components that provide views for data (DataGrid, List, Tree, and so on) can also display Flash Remoting RecordSets 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.

Methods of the DataProvider API

Name	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.
DataProvider.sortItems()	Sorts the items in a data provider.
DataProvider.sortItemsBy()	Sorts the items in a data provider according to a specified compare function.

Properties of the DataProvider API

Name	Description
DataProvider.length	The number of items in a data provider.

Events of the DataProvider API

Name	Description
DataProvider.modelChanged	Broadcast when the data provider is changed.

DataProvider.addItem()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

myDP.addItem(*item*)

Parameters

item An object containing data. This comprises 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDP.addItemAt(index, item)
```

Parameters

index A number greater than or equal to 0. The position at which to insert the item; 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 version 79.

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 in the item to modify.

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 version 79.

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`. 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 allows the data model to 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDP.getItemAt(index)
```

Parameters

index A number greater than or equal to 0 and less than `DataProvider.length`. The index of the item to retrieve.

Returns

A reference to the retreived 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 version 79.

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. This 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.modelChanged

Availability

Flash Player 6 version 79.

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. A listener is typically added to a model by assigning its `dataProvider` property.

V2 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 following are the supported values of the `eventName` property:
  - `updateAll` The entire view needs refreshing, excluding scroll position.
  - `addItems` A series of items have been added.
  - `removeItems` A series of items have been deleted.
  - `updateItems` A series of items need refreshing.
  - `sort` The data has been sorted.
  - `updateField` A field within an item has to be changed and needs refreshing.
  - `updateColumn` An entire field's definition within the dataProvider needs refreshing.
  - `filterModel` The model has been filtered, and the view needs refreshing (reset scrollPosition).
  - `schemaLoaded` The field's definition of the dataProvider 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 about event objects, see "Event Objects" on page 562.

Example

In the following example, a handler called `listener` is defined and passed to the `addEventListener()` method 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, as follows:

```
listener = new Object();
listener.modelChanged = function(evt){
    trace(evt.eventName);
}
myList.addEventListener("modelChanged", listener);
```

DataProvider.length

Availability

Flash Player 6 version 79.

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.removeAll()

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDP.removeItemAt(index)
```

Parameters

index A number greater than or equal to 0. 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDP.replaceItemAt(index, item)
```

Parameters

index A number greater than or equal to 0. 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 with the label "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 is used to compare two items to determine their sort order. For details, see `Array.sort()` in ActionScript Dictionary Help. This parameter is optional.

optionsFlag Allows you to 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 may 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, if sorted with the *optionsFlag* parameter containing the value `Array.RETURNINDEXEDARRAY`, 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 compare function specified by the *compareFunc* parameter or according to one or more of the sort options specified by the *optionsFlag* parameter.

This method triggers the `modelChanged` event with the event name `sort`.

Example

This example sorts based on 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*`, `*order*`, [`*optionsFlag*`])`

Parameters

fieldName A string specifying the name of the field to use for sorting. This value is usually `"label"` or `"data"`.

order A string specifying whether to sort the items in ascending order (`"ASC"`) or descending order (`"DESC"`).

optionsFlag Allows you to 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 may 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, if sorted with the *`optionsFlag`* parameter containing the value `Array.RETURNINDEXEDARRAY`, 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 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. You can optionally use the *`optionsFlag`* parameter to specify a sorting style.

This method triggers the `modelChanged` event with the event name `sort`.

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 representing the fields within a record of data. For details on transfer objects, see Core J2EE Patterns Transfer Object at java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html.

Note: The DataSet component requires Flash Player 7 or later.

Using the DataSet component (Flash Professional only)

You typically use the DataSet component in an application 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.

For more general information about the DataSet component and how to use it with other components, see "Data management (Flash Professional only)" in Using Flash Help.

DataSet component parameters

The following are authoring parameters that you can set for each DataSet component instance in the Property inspector or in the Component Inspector panel:

itemClassName The name of the transfer object class that will be instantiated each time a new item is needed.

Note: To make the specified class available at runtime, you must also make a fully qualified reference to this class somewhere within your SWF file's code (for example, `var myItem:my.package.myItem;`).

filtered If `true`, a filter is applied to the data set so that it contains only the objects that match the filter criteria.

logChanges If `true`, the data set logs all mutations (changes to data or method calls) to its `deltaPacket` property.

readOnly If `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 196.

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 the XMLConnector or WebServiceConnector component. 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).

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 (Window > Development Panels > Components) if it's not already open.
3. Drag a DataSet component from the Components panel to the Stage. In the Property inspector, name it **userData**.
4. Drag a DataGrid component to the Stage and name it **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 name it **nextBtn**.

7 In the Timeline, select the first frame on Layer 1 and open the Actions panel (Window > Development Panels > Actions).

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: `firstName`, `lastName`, and `id`.

9 To bind the contents of the DataSet component to the contents of the DataGrid component, open the Component Inspector panel (Window > Development Panels > Component Inspector) and click the Bindings tab.

10 Select the DataGrid component (`userGrid`) on the Stage, and click the Add Binding (+) button in the Component Inspector panel.

11 In the Add Binding dialog box, select "dataProvider : Array" and click OK.

12 Double-click the Bound To field in the Component Inspector panel.

13 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.

14 To bind the selected index of the DataSet component to the selected index of the DataGrid component, click the Add Binding (+) button again in the Component Inspector panel.

15 In the dialog box that appears, select "selectedIndex : Number". Click OK.

16 Double-click the Bound To field in the Component Inspector panel to open the Bound To dialog box.

17 In the Component Path field, select "DataSet <userData>" from the Component Path column and then select "selectedIndex : Number" from the Schema Location column.

18 Select the Button component (`nextBtn`) and open the Actions panel (Window > Development Panels > Actions), if it is not already open.

19 Enter the following code in the Actions panel:

```
on(click) {
  _parent.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)

ActionScript Class Name mx.data.components.DataSet

Method summary for the DataSet class

Method	Description
DataSet.addItem()	Adds the specified item to the collection.
DataSet.addSort()	Creates a new sorted view of the items in the collection.
DataSet.applyUpdates()	Notifies listeners that changes made to the DataSet object are ready.
DataSet.changesPending()	Indicates whether there are items in the DeltaPacket object.
DataSet.clear()	Clears all items from the current view of the collection.
DataSet.createItem()	Returns a newly initialized collection item.
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.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()	Retrieves the contents of a DataSet object 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.removeRange()	Removes the current iterator's range settings.

Method	Description
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

Property	Description
DataSet.currentItem	Returns the current item in the collection.
DataSet.dataProvider	Returns the DataProvider interface.
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	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 within 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 within the collection.

Event summary for 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.deltaPacketChanged	Broadcast when the DataSet object's delta packet has been changed and is ready to be used.
DataSet.calcFields	Broadcast when calculated fields should be updated.
DataSet.iteratorScrolled	Broadcast when the iterator's position is changed.

Event	Description
DataSet.modelChanged	Broadcast when items in the collection have been modified in some way.
DataSet.newItem	Broadcast when a new item 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 DeltaPacket object 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 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

Returns `true` if the item was added to the collection; otherwise, returns `false`.

Description

Method; adds the specified 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 specified 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

```
myDataSet.addItem(myDataSet.createItem());
```

See also

`DataSet.createItem()`

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 fields 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 (|). The value(s) must be one of the following:

- `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. The new sort is automatically assigned to the current iterator after it is created and stored 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 DataSet `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()
```

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 call 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 panel. 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 Component Inspector panel's Schema tab) is updated.

For more information on the Kind property, see "Schema kinds (Flash Professional only)" in Using Flash Help.

Caution: Do not change the values of any of noncalculated fields in this event, because this will result in an "infinite loop." Only set the values of calculated fields within the `calcFields` event.

DataSet.changesPending()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
dataSet.changesPending()
```

Returns

A Boolean value.

Description

Method; returns `true` if the collection, or any item within the collection, has changes pending that have not yet been sent in a DeltaPacket object; 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 DeltaPacket object.

```
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 within 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 with 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 specified 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 within 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 DataProvider interface for this data set. This property provides data to user interface controls, such as the List and DataGrid components.

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 DeltaPacket object 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 DeltaPacket object. This transaction ID is used to identify the DeltaPacket object on an update round trip from the server and back to the client. Any subsequent assignment to the `deltaPacket` property by a DeltaPacket object with a matching transaction ID is assumed to be the server's response to the changes previously sent. A DeltaPacket object 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 DeltaPacket object with a matching ID is assigned to `DataSet.deltaPacket`. A DeltaPacket object 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 DeltaPacket object.

A `DataSetError` exception is thrown if a DeltaPacket object is assigned with a matching transaction ID and one of the items in the newly assigned DeltaPacket object cannot be found in the original DeltaPacket object.

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 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. When set to `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 those 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 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.

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 those 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 found, the found item 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 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, and that specified in the array. 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, the `DataSet.getItemId()` method is used to get the unique identifier for the item in the collection, and the `DataSet.locateById()` method 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 the addSort() method.
if(studentData.find(["Bobby", 105])) {
  studentID = studentData.getItemId();
}
// Now use the locateByID() method 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, and that specified in the array. For example, if the search value specified is ["05-02-02"], the underlying date field is used to convert the value using 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 the addSort() method.
if(studentData.findFirst(["Bobby", "13"])) {
  studentID = studentData.getItemId();
}
// Now use the locateByID() method 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, and that specified in the array. For example, if the search value specified is `["05-02-02"]`, the underlying date field is used to convert the value using 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, the `DataSet.getItemId()` method is used to get the unique identifier for the item in the collection, and the `DataSet.locateById()` method 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 the addSort() method.
if(studentData.findLast(["Bobby", "13"])) {
  studentID = studentData.getItemId();
}
// Now use the locateByID() method 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 DataSet `userData` 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 of items to get the ID for. 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 within 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 within 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.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 the `DataSet.useSort()` method. If a sort by that name doesn't exist, one is created by means of the `DataSet.addSort()` method.

```
if(myDataSet.hasSort("customerSort"))
  myDataSet.useSort("customerSort");
} else {
  myDataSet.addSort("customerSort", ["customer"],
  DataSetIterator.Descending);
}
```

See also

`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 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 within 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 about the TransferObject interface, see "TransferObject interface" on page 527.

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 within this DataSet collection. Note that the instance name of the DataSet collection is used to identify the data within 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, the `DataSet.getItemId()` method is used to get the unique identifier for that item, and the `DataSet.locateById()` method 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 within the data provider needs refreshing.
`"updateField"`	A field within 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` will prevent updates to the collection.

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 priviledges.
    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 that should 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.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 means of `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 a DeltaPacket object is assigned to `DataSet.deltaPacket` whose transaction ID matches that of a DeltaPacket object previously retrieved from the DataSet object, and that has messages associated with any of the Delta or DeltaItem objects contained by that DeltaPacket object.

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 resent.

The event object (*eventObj*) contains the following properties:

`target` The DataSet object that generated the event.

`type` The string `"resolveDelta"`.

`data` An array of Delta 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">
      <options/>
    </kind>
  </property>
  <property> ... </property>
  ...
</properties>
```

A `DataSetError` exception is thrown if the XML specified does not follow the above format.

Example

```
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 within 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 194.

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 it is being assigned to; 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 within 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 sort specified by *sortName* does not exist, a DataSetError exception is thrown.

To create a sort, use the 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 on a date to select it. You can set parameters that indicate the month and day names, the first day of the week, and any disabled dates, as well as highlighting the currrent date.

A live preview of each DateChooser instance reflects the values indicated by the Property inspector or Component Inspector panel while 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

The following are authoring parameters that you can set for each DateChooser component instance in the Property inspector or in the Component Inspector panel:

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 DateChooser. 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 7 values. The default value is [] (an empty array).

showToday indicates whether or not 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 239.

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 DateChooser 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, do the following:

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 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 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 Control > Test Movie.

Customizing the DateChooser component (Flash Professional only)

You can transform a DateChooser component horizontally and vertically both 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 date chooser instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

A DateChooser component supports the following Halo styles:

Style	Description
`themeColor`	The glow color for the rollover and selected dates. This is the only color style that doesn't inherit its value. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`.
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.

Style	Description
fontSize	The point size for the font.
fontStyle	The font style; either "normal", or "italic".
fontWeight	The font weight; either "normal", or "bold".
textDecoration	The text decoration: either "none", or "underline".

Using skins with the DateChooser component

The DateChooser component skins to represent its visual states. To skin the DateChooser component while authoring, modify skin symbols in the Flash UI Components 2/Themes/MMDefault/DateChooser Assets/Elements/Month skins states folder in the library of one of the themes FLA files. For more information, see "About skinning components" on page 36.

Only the month scrolling buttons can be dynamically skinned in this component. A DateChooser component uses the following skin properties:

Property	Description
falseUpSkin	The up state. The default values are fwdMonthUp and backMonthUp.
falseDownSkin	The down state. The default values are fwdMonthDown and backMonthDown.
falseDisabledSkin	The disabled state. The default values are fwdMonthDisabled and backMonthDisabled.

DateChooser class (Flash Professional only)

Inheritance UIObject > UIComponent > DateChooser

ActionScript Class Name mx.controls.DateChooser

The properties of the DateChooser class allow you to 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 panel.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.DateChooser.version);
```

Note: The following code returns undefined: `trace(myDC.version);`.

Method summary for the DateChooser class

Inherits all methods from UIObject and UIComponent.

Property summary for 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.
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.

Inherits all properties from UIObject and UIComponent.

Event summary for the DateChooser class

Event	Description
DateChooser.change	Broadcast when a date is selected.
DateChooser.scroll	Broadcast when the month buttons are pressed.

Inherits all events from UIObject and UIComponent.

DateChooser.change

Availability

Flash Player 6 version 79.

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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a DateChooser 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()` action 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 the `UIEventDispatcher.addEventListener()` method from `myDC` and passes it the `change` event and the `form` listener object as parameters, as in the following:

```
form.change = function(eventObj){
    trace("date selected " +  eventObj.target.selectedDate) ;
}
myDC.addEventListener("change", form);
```

DateChooser.dayNames

Availability

Flash Player 6 version 79.

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 5th day of the week (Thursday) from "T" to "R":

```
myDC.dayNames[4] = "R";
```

DateChooser.disabledDays

Availability

Flash Player 6 version 79.

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 between 0 (Sunday) and 6 (Saturday). The default value is `[]` (empty array).

Example

The following example disables Sundays and Saturdays so that users can only select weekdays:

```
myDC.disabledDays = [0, 6];
```

DateChooser.disabledRanges

Availability

Flash Player 6 version 79.

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 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:

```
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 version 79.

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 version 79.

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 version 79.

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 version 79.

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 version 79.

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 pressed.

The first usage example uses an `on()` handler and must be attached directly to a DateChooser 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 stepper `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 a set of properties that contains 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 `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a month button is pressed 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` action 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 the `UIEventDispatcher.addEventListener()` method from `myDC` and passes it the `scroll` event and the `form` listener object as parameters, as in the following:

```
form = new Object();
form.scroll = function(eventObj){
  trace(eventObj.detail);
}
myDC.addEventListener("scroll", form);
```

DateChooser.selectableRange

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myDC.selectableRange
```

Description

Property; sets a single selectable date or a range of selectable dates. The user will not be able to 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 value 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 - July, 2003, the displayed month will change to July, 2003.

Example

The following example defines the selectable range to 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 to the dates after and including May 7:

```
myDC.selectableRange = {rangeStart: new Date(2003, 4, 7)};
```

The following example defines the selectable range to 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 version 79.

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.

The `selectedDate` property cannot be set inside a `disabledRange`, outside a `selectableRange`, or on a day that has been disabled. If the `selectedDate` property is set to one of the previous dates, the value will be 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 version 79.

Edition

Flash MX Professional 2004.

Usage

myDC`.showToday`

Description

Property; this property 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 panel while authoring because it is a pop-up component that is not visible while 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

The following are authoring parameters that you can set for each DateField component instance in the Property inspector or in the Component Inspector panel:

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 DateChooser. 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 7 values. The default value is [] (an empty array).

showToday indicates whether or not 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 251.

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, do the following:

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 both 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 within 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 non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

A DateField component supports the following Halo styles:

Style	Description
`themeColor`	The glow color for the rollover and selected dates. This is the only color style that doesn't inherit its value. Possible values are `"haloGreen"`, `"haloBlue"`, and `"haloOrange"`.
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style; either `"normal"`, or `"italic"`.
`fontWeight`	The font weight; either `"normal"`, or `"bold"`.
`textDecoration`	The text decoration: either `"none"`, or `"underline"`.

Using skins with the DateField component

The DateField component uses skins to represent the visual states of the pop-up icon. To skin the pop-up icon while authoring, modify skin symbols in the Flash UI Components 2/Themes/MMDefault/DateField Elements skins states folder in the library of one of the themes FLA files. For more information, see "About skinning components" on page 36.

Only the pop-up icon button can be skinned in this component. 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.

DateField class (Flash Professional only)

Inheritance UIObject > UIComponent > ComboBase > DateField

ActionScript Class Name mx.controls.DateField

The properties of the DateField class allow you to 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 panel.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.DateField.version);
```

Note: The following code returns undefined: `trace(myDateFieldInstance.version);`.

Method summary for the DateField class

Method	Description
`DateField.close()`	Closes the pop-up date chooser subcomponent.
`DateField.open()`	Opens the pop-up date chooser subcomponent.

Inherits all methods from UIObject and UIComponent.

Property summary for 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 days of the week that are disabled for all applicable dates in the date chooser.
`DateField.disabledRanges`	A range of disabled dates or a single disabled date.
`DateField.displayedMonth`	A number indicating an element in the `monthNames` array to display in the date chooser.
`DateField.displayedYear`	A number indicating the year to display.
`DateField.firstDayOfWeek`	A number indicating an element in the `dayNames` array to display in the first column of the date chooser.
`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.

Property	Description
DateField.selectedDate	A Date object indicating the selected date.
DateField.showToday	A Boolean value indicating whether the current date is highlighted.

Inherits all properties from UIObject and UIComponent.

Event summary for the DateField class

Event	Description
DateField.change	Broadcast when a date is selected.
DateField.close	Broadcast when the date chooser subcomponent closes.
DateField.open	Broadcast when the date chooser subcomponent opens.
DateField.scroll	Broadcast when the month buttons are pressed.

Inherits all events from UIObject and UIComponent.

DateField.change

Availability

Flash Player 6 version 79.

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 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 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 (*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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 action 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 the UIEventDispatcher.addEventListener() method from myDF and passes it the change event and the form listener object as parameters, as in the following:

```
form.change = function(eventObj){
    trace("date selected " +  eventObj.target.selectedDate) ;
}
myDF.addEventListener("change", form);
```

DateField.close()

Availability

Flash Player 6 version 79.

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 version 79.

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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when the date chooser within `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` action 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 the `UIEventDispatcher.addEventListener()` method from `myDF` and passes it the `close` event and the `form` listener object as parameters, as in the following:

```
form.close = function(eventObj){
  trace("PullDown Closed" + eventObj.target.selectedDate);
}
myDF.addEventListener("close", form);
```

DateField.dateFormatter

Availability

Flash Player 6 version 79.

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 version 79.

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 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 5th day of the week (Thursday) from "T" to "R":

```
myDF.dayNames[4] = "R";
```

DateField.disabledDays

Availability

Flash Player 6 version 79.

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 `[]` (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 version 79.

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 version 79.

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 version 79.

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 version 79.

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 `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 version 79.

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 version 79.

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 version 79.

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 date chooser 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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a stepper 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` action 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 the `UIEventDispatcher.addEventListener()` method from `myDF` and passes it the `open` event and the `form` listener object as parameters, as in the following:

```
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 version 79.

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 version 79.

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 pressed.

The first usage example uses an `on()` handler and must be attached directly to a DateField 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 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 a set of properties that contains 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 `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a month button is pressed 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` action 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 the `UIEventDispatcher.addEventListener()` method from `myDateField` and passes it the `scroll` event and the `form` listener object as parameters, as in the following:

```
form = new Object();
form.scroll = function(eventObj){
  trace(eventObj.detail);
}
myDF.addEventListener("scroll", form);
```

DateField.selectableRange

Availability

Flash Player 6 version 79.

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 value of `DateField.displayedMonth` and `DateField.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 - July, 2003, the displayed month will change to July, 2003.

Example

The following example defines the selectable range to 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 to the dates after and including May 7:

```
myDF.selectableRange = {rangeStart: new Date(2003, 4, 7)};
```

The following example defines the selectable range to 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 version 79.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

myDF.showToday

Description

Property; this property 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;
```

DepthManager class

ActionScript Class Name `mx.managers.DepthManager`

The DepthManager class adds functionality to the ActionScript MovieClip class that allows you to manage the relative depth assignments of any component or movie clip, including `_root`. It also allows you to manage reserved depths in a special highest-depth clip on the `_root` for system-level services like the cursor or tooltips.

The following methods compose the relative depth-ordering API:

- `DepthManager.createChildAtDepth()`
- `DepthManager.createClassChildAtDepth()`
- `DepthManager.setDepthAbove()`
- `DepthManager.setDepthBelow()`
- `DepthManager.setDepthTo()`

The following methods compose the reserved depth space API:

- `DepthManager.createClassObjectAtDepth()`
- `DepthManager.createObjectAtDepth()`

Method summary for 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 that 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.

DepthManager.createChildAtDepth()

Availability

Flash Player 6 version 79.

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.

Description

Method; creates a child instance of the symbol specified by the *linkageName* parameter at the depth specified by the *depthFlag* parameter.

Example

The following example creates a minuteHand instance of the MinuteSymbol movie clip and places it on top of the clock:

```
import mx.managers.DepthManager;
minuteHand = clock.createChildAtDepth("MinuteSymbol", DepthManager.kTop);
```

DepthManager.createClassChildAtDepth()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004 and Flash MX Professional 2004

Usage

movieClipInstance.createClassChildAtDepth(*className*, *depthFlag*[, *initObj*])

Parameters

className A class name.

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.

Description

Method; creates a child of the class specified by the *className* parameter at the depth specified by the *depthFlag* parameter.

Example

The following code draws a focus rectangle on top 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 version 79.

Edition

Flash MX 2004 and Flash MX Professional 2004

Usage

```
DepthManager.createClassObjectAtDepth(className, depthSpace[, initObj])
```

Parameters

className A class name.

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.

Description

Method; creates an object of the class specified by the *className* parameter at the depth specified by the *depthSpace* parameter. 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 version 79.

Edition

Flash MX 2004 and Flash MX Professional 2004

Usage

```
DepthManager.createObjectAtDepth(linkageName, depthSpace[, initObj])
```

Parameters

linkageName A linkage identifier.

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.

Returns

A reference to the created object.

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.setDepthAbove()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004 and Flash MX Professional 2004

Usage

```
movieClipInstance.setDepthAbove(instance)
```

Parameters

instance An instance name.

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.

DepthManager.setDepthBelow()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004 and Flash MX Professional 2004

Usage

```
movieClipInstance.setDepthBelow(instance)
```

Parameters

instance An instance name.

Returns

Nothing.

Description

Method; sets the depth of a movie clip or component instance below the depth of the instance specified by the *instance* parameter.

Example

The following code sets the depth of the `textInput` instance below the depth of the `button`:

```
textInput.setDepthBelow(button);
```

DepthManager.setDepthTo()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004 and Flash MX Professional 2004

Usage

```
movieClipInstance.setDepthTo(depth)
```

Parameters

depth A depth level.

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 ActionScript Reference Guide Help.

FocusManager class

You can use the FocusManager to specify the order in which components receive focus when a user presses the Tab key to navigate in an application. You can use the FocusManager API to set a button in your document that receives keyboard input when a user presses Enter (Windows) or Return (Macintosh). For example, when a user fills 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 FocusManager support; you don't need to write code to invoke it. The FocusManager also 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 a FocusManager so the components in that window become their own tab set, preventing the user from tabbing into components in other windows.

The FocusManager 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 groupName as the current object. If it does, it automatically moves focus to the next object with a different groupName. Other sets of components that support a groupName property can also use this feature.

The FocusManager handles focus changes due to mouse clicks. If the user clicks on a component, that component is given focus.

The FocusManager does not automatically assign focus to a component in an application. The main window and any pop-up windows will not have focus set on any component by default unless you call FocusManager.setFocus() on a component.

Using the FocusManager

The FocusManager 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.

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 FocusManager looks for an enabled object with a tabIndex property that is higher than the current value of tabIndex. Once the FocusManager 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 FocusManager uses the z-order. The z-order is set up primarily by the order components are dragged to the Stage, however, you can also use the Modify/Arrage/Bring-to-Front/Back commands to determine the final z-order.

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 in the following:

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

Note: The FocusManager 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.

FocusManager parameters

There are no authoring parameters for the FocusManager. You must use the ActionScript methods and properties of the FocusManager class in the Actions panel. For more information, see FocusManager class.

Creating an application with the FocusManager

The following procedure creates a focus scheme in a Flash application.

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);
focusManager.defaultPushButton = okButton;
lo = new Object();
lo.click = function(evt){
  trace(evt.target + " was clicked");
}
okButton.addEventListener("click", lo);
```

This code sets the tab ordering and specifies a default button to receive a `click` event when a user presses Enter (Windows) or Return (Macintosh).

Customizing the FocusManager

You can change the color of the focus ring in the Halo theme by changing the value of the `themeColor` style.

The FocusManager 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

Inheritance UIObject > UIComponent > FocusManager

ActionScript Class Name mx.managers.FocusManager

Method summary for 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.

Property summary for the FocusManager class

Method	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.

FocusManager.defaultPushButton

Availability

Flash Player 6 version 79.

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 the Enter key (Windows) or Return key (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 FocusManager 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 is currently focused 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 version 79.

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 ActionScript Dictionary Help) or `focusOut` event. The default value is `true`.

Example

The following code disables default push button handling:

```
focusManager.defaultPushButtonEnabled = false;
```

FocusManager.enabled

Availability

Flash Player 6 version 79.

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 FocusManager.) Setting `enabled` to `false` allows a component to receive the tab handling keys and handle them internally. You must re-enable the FocusManager handling by watching the component's `onKillFocus()` method (see `MovieClip.onKillFocus` in ActionScript Dictionary Help) or `focusOut` event. The default value is `true`.

Example

The following code disables tabbing:

```
focusManager.enabled = false;
```

FocusManager.getFocus()

Availability

Flash Player 6 version 79.

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 currently focused object is `myInputText`:

```
if (focusManager.getFocus() == myInputText)
{
  focusManager.setFocus(myOKButton);
}
```

See also

`FocusManager.setFocus()`

FocusManager.nextTabIndex

Availability

Flash Player 6 version 79.

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 version 79.

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.sendDefaultPushButtonEvent()

FocusManager.setFocus()

Availability

Flash Player 6 version 79.

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.

Example

The following code sets focus to `myOKButton`:

```
focusManager.setFocus(myOKButton);
```

See also

FocusManager.getFocus()

Form class (Flash Professional only)

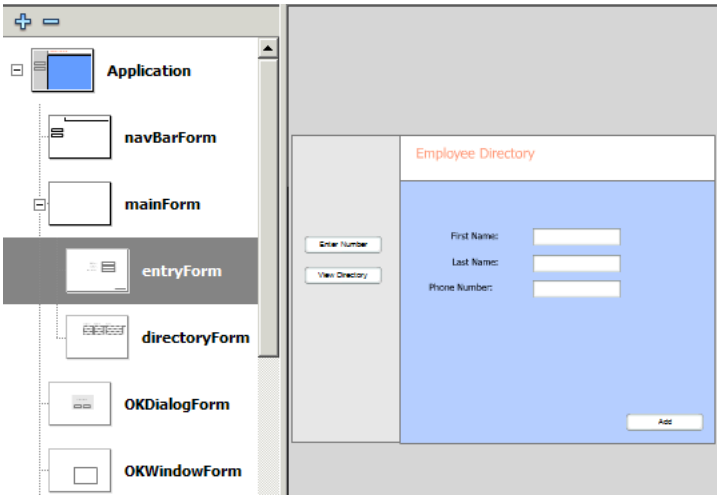
Inheritance UIObject > UIComponent > View > Loader > Screen > Form

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. For an overview of working with screens, see "Working with Screens (Flash Professional Only)" in Using Flash Help.

Using the Form class (Flash Professional only)

Forms function as both 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 (Window > Development Panels > Behaviors) 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 Help.

Because the Form class extends the Loader class, you can easily load external content (either a SWF or JPEG) into a form. For example, the contents of a form could be a separate SWF, which itself might contain forms. In this way, you can modularize your form applications, 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 452.

Form object parameters

The following are authoring parameters that you can set for each Form object instance in the Property inspector or in the Component Inspector panel:

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 JPG file to load into the slide. By default, loaded content clips to fit the slide.

visible Specifies whether the form is visible (true) or not (false) when it first loads.

Method summary for the Form class

Method	Description
`Form.getChildForm()`	Returns the child form at a specified index.

Inherits all methods from UIObject, UIComponent, View, Loader component, and Screen class (Flash Professional only).

Property summary for the Form class

Property	Description
`Form.currentFocusedForm`	Returns the "leafmost" form that contains the global current focus.
`Form.indexInParentForm`	Returns the index (zero-based) of this form in its parent's list of subforms.
`Form.visible`	Specifies whether the form is visible when its parent form, slide, movie clip, or SWF is visible.
`Form.numChildForms`	Returns the number of child forms that this form contains.
`Form.parentIsForm`	Returns whether or not the parent object of this form is also a form.
`Form.rootForm`	Returns the root of the form tree, or subtree, that contains the form.

Inherits all properties from UIObject, UIComponent, View, Loader component, and Screen class (Flash Professional only).

Form.currentFocusedForm

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myForm.getChildForm(childIndex)
```

Parameters

childIndex A number that indicates the index (zero-based) 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 version 79.

Edition

Flash MX Professional 2004.

Usage

myForm.indexInParentForm

Description

Property (read-only); contains the index (zero-based) 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, it is a slide), then 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 version 79.

Edition

Flash MX Professional 2004.

Usage

myForm.numChildForms

Description

Property (read-only); returns the number of child forms contained by *myForm*. This property does not includes any slides that are contained my *myForm*, only forms.

Example

The following code iterates over all the child forms contained my `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 version 79.

Edition

Flash MX Professional 2004.

Usage

`myForm``.parentIsForm`

Description

Property (read-only): returns a Boolean (`true` or `false`) value indicating whether the specified form's parent object is also a form (`true`), or not (`false`). If false, then *`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.rootForm

Availability

Flash Player 6 version 79.

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), then this property returns *`myForm`*.

Example

In the following example, a reference to the root form of `myForm` is placed into 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:my.screens.Form = myForm.rootForm;
if(rootForm == myForm) {
  trace("myForm is the top form in its tree");
}
```

Form.visible

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

myForm.visible

Description

Property; determines whether *myForm* is visible when its parent form, slide, movie clip, or movie 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 Help.

Example

The following code, attached to an instance of the Button component, sets to `false` the `visible` property of the form that contains the button.

```
on(click) {
  _parent.visible = true;
}
```

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 alignment and sizing 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 panel while 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 only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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 (v2) 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.

Label parameters

The following are authoring parameters that you can set for each Label component instance in the Property inspector or in the Component Inspector panel:

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 the `html` 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 be any of the following four values:

- 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 the its original horizontal center position.
- right—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 `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.

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, do the following:

1. Drag a Label component from the Components panel to the Stage.
2. In the Component Inspector panel, enter **Expiration Date** for the label parameter.

Customizing the label component

You can transform a Label component horizontally and vertically both 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 283. 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 non-color style properties.

For more information about styles, see "Using styles to customize component color and text" on page 27.

A Label component supports the following styles:

Style	Description
`color`	The default color for text.
`embedFonts`	The fonts to embed in the document.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style, either "normal",or "italic".
`fontWeight`	The font weight, either "normal" or "bold".
`textAlign`	The text alignment: either "left", "right", or "center".
`textDecoration`	The text decoration, either "none" or "underline".

Using skins with the Label component

The Label component is not skinnable.

For more information about skinning a component, see "About skinning components" on page 36.

Label class

Inheritance UIObject > 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 panel.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.Label.version);
```

Note: The following code returns undefined: `trace(myLabelInstance.version);`.

Method summary for the Label class

Inherits all methods from UIObject.

Property summary for 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.

Inherits all properties from UIObject.

Event summary for the Label class

Inherits all events from UIObject.

Label.autoSize

Availability

Flash Player 6 version 79.

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 the its original horizontal center position.
- `right`—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 `autoSize` property is different from the built-in ActionScript TextField object's `autoSize` property.

Label.html

Availability

Flash Player 6 version 79.

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.

You cannot use the `<font color>` HTML tag with the Label component even when `Label.html` is set to `true`. For example, in the following example, the text "Hello" displays black, not red as it would if `<font color>` were supported:

```
lbl.html = true;
lbl.text = "<font color=\"#FF0000\">Hello</font> World";
```

In order 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 off-screen 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, as follows:

```
labelControl.html = true;
labelControl.text = "The <b>Royal</b> Nonesuch";
```

The word "Royal" displays in bold.

Label.text

Availability

Flash Player 6 version 79.

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 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
Alphanumerical keys	Jump to the next item with Key.getAscii() as the first character in its label.
Control	Toggle key. Allows multiple non-contiguous selects and deselects.
Down	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	Contiguous selection key. Allows for contiguous selection.
Up	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 24 or "FocusManager class" on page 270.

A live preview of each List instance on the Stage reflects changes made to parameters in the Property inspector or Component Inspector panel while 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 only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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.

List component parameters

The following are authoring parameters that you can set for each List component instance in the Property inspector or in the Component Inspector panel:

data 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 An array of text values that populate the label values of list. The default value is [] (an empty array). There is no equivalent runtime property.

multipleSelection A Boolean value that indicates whether you can select multiple values (true) or not (false). The default value is false.

rowHeight This 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.

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, do the following:

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.

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 List component to an application, do the following:

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 both 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 464.

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 Halo styles:

Style	Description
`alternatingRowColors`	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.
`backgroundColor`	The background color of the list. This style is defined on a class style declaration, ScrollSelectList.
`borderColor`	The black section of a three-dimensional border or the color section of a two-dimensional border.
`borderStyle`	The bounding box style. The possible values are: "none", "solid", "inset" and "outset". This style is defined on a class style declaration, ScrollSelectList.
`defaultIcon`	Name of the default icon to use for list rows. The default value is undefined.
`rollOverColor`	The color of a rolled over row.
`selectionColor`	The color of a selected row.
`selectionEasing`	A reference to an easing equation (function) used for controlling programmatic tweening.

Style	Description
disabledColor	The disabled color for text.
textRollOverColor	The color of text when the pointer rolls over it.
textSelectedColor	The color of text when selected.
selectionDisabledColor	The color of a row if it has been selected and disabled.
selectionDuration	The length of any transitions when selecting items.
useRollOver	Determines whether rolling over a row activates highlighting.

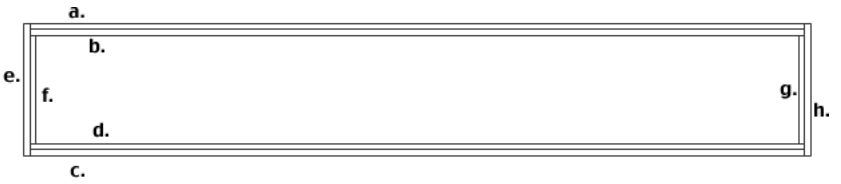
A List component also uses the style properties of the Label component (see "Using styles with the Label component" on page 284), the ScrollPane component (see "ScrollPane component" on page 464), and RectBorder.

Using skins with the List component

All the skins in the List component are included in the subcomponents from which the list is composed (ScrollPane component and RectBorder). For more information, see "ScrollPane component" on page 464. You can use the `setStyle()` method (see `UIObject.setStyle()`) to change the following RectBorder style properties:

RectBorder styles	Border position
borderColor	a
highlightColor	b
borderColor	c
shadowColor	d
borderCapColor	e
shadowCapColor	f
shadowCapColor	g
borderCapColor	h

The style properties set the following positions on the border:



List class

Inheritance UIObject > UIComponent > View > ScrollView > ScrollSelectList > List

ActionScript Class Name mx.controls.List

The List component is composed of three parts:

- Items
- Rows
- 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 interface, 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 allow you to 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, ComboBoxes, DataGrids, and so on. The List component has a set of 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 FocusManager 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 24.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.List.version);
```

Note: The following code returns undefined: `trace(myListInstance.version);`.

Method summary for 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.

Method	Description
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.

Inherits all methods from UIObject and UIComponent.

Property summary for 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 within 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 length of the list in items. This property is read-only.
List.maxHPosition	Specifies 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").

Inherits all properties from UIObject and UIComponent.

Event summary for the List class

Event	Description
`List.change`	Broadcast whenever the selection changes due to user interaction.
`List.itemRollOut`	Broadcast when list items are rolled over and then off by the pointer.
`List.itemRollOver`	Broadcast when list items are rolled over by the pointer.
`List.scroll`	Broadcast when a list is scrolled.

Inherits all events from UIObject and UIComponent.

List.addItem()

Availability

Flash Player 6 version 79.

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 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 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 version 79.

Edition

Flash MX 2004.

Usage

```
listInstance.addItemAt(index, label[, data])
listInstance.addItemAt(index, 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 any data type.

index A number greater than or equal to zero that indicates the position of the item.

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 version 79.

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" on page 77.

Example

The following example uses a linkage identifier to set a new cell renderer:

```
myList.cellRenderer = "ComboBoxCell";
```

List.change

Availability

Flash Player 6 version 79.

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 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 list component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

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

`UIEventDispatcher.addEventListener()`

List.dataProvider

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`listnstance``.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 interface. The default value is []. For more information about the DataProvider interface, see "DataProvider API" on page 183.

The List component, and other data-aware components, add methods to the Array object's prototype so that they conform to the DataProvider interface. 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 is 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 lose the selection.

Any instance that implements the DataProvider interface can be a data provider for a List component. This includes Flash Remoting RecordSets, Firefly DataSets, 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 DataProvider will be broadcast to the list
  myDP.addItem({ label: accounts[i].name,
                 data:  accounts[i].accountID });
}
```

List.getItemAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`listInstance``.getItemAt(`*`index`*`)`

Parameters

`index` A number greater than or equal to 0, and less than `List.length`. The index of the item to retrieve.

Returns

The indexed item object. Undefined if index is out of range.

Description

Method; retrieves the item at a 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 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

`listInstance``.hScrollPolicy`

Description

Property; a string that determines whether or not 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: The value `"auto"` is not supported for `List.hScrollPolicy`.

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 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

`listInstance``.iconFunction`

Description

Property; specifies a function to be used to determine 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 version 79.

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 additional property: `index`. The `index` is the number of the item that was rolled out.

Description

Event; broadcast to all registered listeners when the list items are rolled out.

The first usage example uses an `on()` handler and must be attached directly to a List 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 additional property: `index`. The `index` is 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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

Edition

Flash MX 2004.

Usage

`listInstance``.labelField`

Description

Property; specifies a field within 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 version 79.

Edition

Flash MX 2004.

Usage

`listInstance``.labelFunction`

Description

Property; specifies a function to be used to decide which field (or field combination) to display of each item. 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 version 79.

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 version 79.

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 will 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 version 79.

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 may be selected:

```
if (myList.multipleSelection){
  // your code here
}
```

The following example allows the list to take multiple selections:

```
myList.selectMultiple = true;
```

List.removeAll()

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

```
listInstance.removeItemAt(index)
```

Parameters

index A string that indicates the label for the new item. A value greater than zero 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 index indicated by the *index* parameter 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 version 79.

Edition

Flash MX 2004.

Usage

```
listInstance.replaceItemAt(index, label[, data])
listInstance.replaceItemAt(index, itemObject)
```

Parameters

index A number greater than zero 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 index specified by the *index* parameter.

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 version 79.

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 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 while 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 version 79.

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 version 79.

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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 version 79.

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.

Example

This example selects the item after the currently selected item. If nothing is selected, item 0 is selected, as follows:

```
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 version 79.

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 0-length array (or undefined) clears the current selection. The value is undefined if nothing is selected.

The `selectedIndices` property is listed in the order that 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 gets 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 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

listInstance.selectedItems

Description

Property (read-only); an array of the selected item objects. In a multiple-selection list, selectedItems allows you to access the set of items selected as item objects.

Example

The following example gets an array of selected item objects:

```
var myObjArray = myList.selectedItems;
```

See also

List.selectedIndex, List.selectedItem, List.selectedIndices

List.setPropertiesAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

listInstance.setPropertiesAt(*index*, *styleObj*)

Parameters

index A number greater than zero 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 properties specified by the *styleObj* parameter to the item specified by the *index* parameter. 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 version 79.

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 ActionScript Dictionary Help.

Returns

The index at which the item was added.

Description

Method; sorts the items in the list according to the *compareFunc* parameter.

Example

The following example sorts the items based on 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 version 79.

Edition

Flash MX 2004.

Usage

```
listInstance.sortItemsBy(fieldName, order)
```

Parameters

fieldName A string that specifies the name of the property to be used for sorting. Typically, this value is "label" or "data".

order A string that specifies whether to sort the items in ascending order ("ASC") or descending order ("DESC").

Returns

Nothing.

Description

Method; sorts the items in the list alphabetically or numerically, in the specified order, using the *fieldName* specified. 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.

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 version 79.

Edition

Flash MX 2004.

Usage

listInstance.vPosition

Description

Property; scrolls the list so that index is the topmost visible item. If index is out of bounds, goes to the nearest in-bounds 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 version 79.

Edition

Flash MX 2004.

Usage

```
listInstance.vScrollPolicy
```

Description

Property; a string that determines whether or not the list supports vertical scrolling. This property can be one of the following values: `"on"`, `"off"` or `"auto"`. The value `"auto"` causes a scroll bar to appear when its 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 a JPEG. 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 24 or "FocusManager class" on page 270.

A live preview of each Loader instance reflects changes made to parameters in the Property inspector or Component Inspector panel while authoring.

Content that is loaded into a Loader component may be enabled for accessibility. If so, you can use the Accessibility panel to make it accessible to screen readers. For more information, see "Creating Accessible Content" in Using Flash Help.

Using the Loader component

You can use a loader whenever you need to grab 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.

Loader component parameters

The following are authoring parameters that you can set for each Loader component instance in the Property inspector or in the Component Inspector panel:

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 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 for testing 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 had started.

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.

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, do the following:

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 panel 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 both 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 = true`, the content is scaled to fit within the bounds of the loader (and is rescaled when `UIObject.setSize()` is called). When the property is `scaleContent = false`, the size of the component is fixed to the size of the content and the `UIObject.setSize()` method has no effect.

Using styles with the Loader component

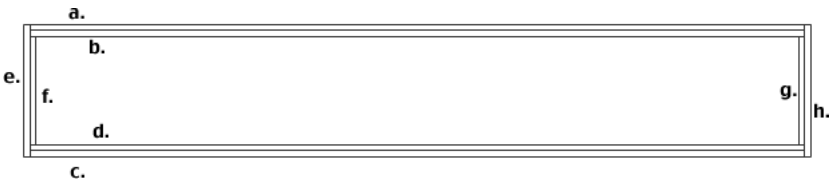
The Loader component doesn't use styles.

Using skins with the Loader component

The Loader component uses RectBorder which uses the ActionScript Drawing API. You can use the `setStyle()` method (see `UIObject.setStyle()`) to change the following RectBorder style properties:

RectBorder styles	Letter
`borderColor`	a
`highlightColor`	b
`borderColor`	c
`shadowColor`	d
`borderCapColor`	e
`shadowCapColor`	f
`shadowCapColor`	g
`borderCapColor`	h

The style properties set the following positions on the border:



Loader class

Inheritance UIObject > UIComponent > View > Loader

ActionScript Class Name mx.controls.Loader

The properties of the Loader class allow you to 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 panel.

For more information, see "Creating custom focus navigation" on page 24.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.Loader.version);
```

Note: The following code returns undefined: `trace(myLoaderInstance.version);`.

Method summary for the Loader class

Method	Description
`Loader.load()`	Loads the content specified by the contentPath property.

Inherits all methods from UIObject and UIComponent.

Property summary for the Loader class

Property	Description
`Loader.autoLoad`	A Boolean value that indicates whether the content loads automatically (`true`) or if 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 specified by the `Loader.contentPath` property. 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`).

Inherits all properties from UIObject and UIComponent.

Event summary for the Loader class

Event	Description
`Loader.complete`	Triggered when the content finished loading.
`Loader.progress`	Triggered while content is loading.

Inherits all properties from UIObject and UIComponent

Loader.autoLoad

Availability

Flash Player 6 version 79.

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 version 79.

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 ProgressBar 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, 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 which generated the change event,
  // i.e., the Loader.
  pBar.setProgress(loader.bytesLoaded, loader.bytesTotal); // show progress
}
loader.addEventListener("progress", loadListener);
loader.content = "logo.swf";
```

When you create an instance with the `createClassObject()` method, you have to position it on Stage with the `move()` and `setSize()` methods. See `UIObject.move()` and `UIObject.setSize()`.

See also

`Loader.bytesTotal`, `UIObject.createClassObject()`

Loader.bytesTotal

Availability

Flash Player 6 version 79.

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 ProgressBar 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 which generated the change event,
  // i.e., 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 version 79.

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 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 Loader component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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);S
loader.contentPath = "logo.swf";
```

Loader.content

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

loaderInstance.content

Description

Property (read-only); a reference to the content of the loader. The value is undefined until the load begins.

See also

Loader.contentPath

Loader.contentPath

Availability

Flash Player 6 version 79.

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 that loads the content. The URL must be in the same subdomain as the URL as the loading SWF.

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 version 79.

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 for the `load()` method to begin loading content. It then calls `load()` and indicates the content to load:

```
createClassObject(mx.controls.Loader, "loader", 0);
loader.autoload = false;
loader.load("logo.swf");
```

Loader.percentLoaded

Availability

Flash Player 6 version 79.

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 a 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 which generated the change event,
  // i.e., 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 version 79.

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 is triggered when the load is triggered by the autoload parameter or by a call to `Loader.load()`. 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.

The first usage example uses an `on()` handler and must be attached directly to a Loader 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 Loader component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 about 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 which generated the change event,
  // i.e., 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 version 79.

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.strechContent = false;
```

Media components (Flash Professional only)

The streaming media components make it easy to incorporate streaming media into Flash presentations. These components allow you to present your media in a variety of ways.

The following are the three media components available to you:

- The MediaDisplay component allows media to be streamed into your Flash content without a supporting user interface. This component can be used with video and audio data. The user of your application will have no control over the media when the MediaDisplay component is used by itself.
- The MediaController component compliments the MediaDisplay component by providing a user interface that controls media playback using standard controls (play, pause, and so on). Media is never loaded into or played by the MediaController; it is used only for controlling playback in a MediaPlayback or MediaDisplay instance. The MediaController component features a "drawer," which exposes 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 7 or later.
- Scan forward and scan backward functionality is not supported by the media components. However, you can achieve this functionality by moving the playback slider.
- Only component size and controller policy are reflected in Live Preview.
- The media components do not support accessibility.

Interacting with media components (Flash Professional only)

The streaming MediaPlayback and MediaController components respond to both mouse and keyboard activity. The MediaDisplay component does not respond to keyboard or mouse events. 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 over	Button is highlighted.
Playback controls of a given controller	Single click of left mouse button	Users can manipulate the playback of audio and video media through the playback controls for a given controller by clicking the playback controls to trigger their corresponding effects. The Pause/Play and Go to Beginning/Go to End buttons follow the standard button behaviors. 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.
Slider controls of a given controller	Move slider back and forth	The playback slider indicates the user's position within the media content; the display handle moves horizontally (by default) to indicate the playback from beginning (left) to end (right). The playback slider moves from bottom to top when the controls are oriented vertically. As the indicator handle 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 indicator handle remains unhighlighted until the indicator passes. Users can drag the indicator handle to affect the media content's playback position. Media begins automatic playback from the point at which the mouse is released if media is playing. If the media is paused, the slider can be moved and released and the media will remain paused. There is also a volume slider, which can be moved from left (muted) to right (maximum volume) in both the horizontal and vertical layouts.

Target	Navigation	Description
Playback controller navigation	Tab, Shift+Tab	Moves the focus from button to button within the controller component, where the focused element will become 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 via 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 control provides focus to the next control in the tab index on the Stage.
A given control button	Space 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)

Before you start using media components, it is a good idea to understand how they work. This section provides an overview of how the media components work. Most of the properties listed in this section can be directly set with the Component Inspector panel. See "Using the Component Inspector panel with media components" on page 332.

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 and should be set such that its name has 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. Cue points can be subsequently removed by means of `Media.removeCuePoint()` and `Media.removeAllCuePoints()`.

The streaming media components broadcast a number of 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 Pause/Play button is clicked. 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 volume controls are adjusted by the user. (See `Media.volume`.)
- A `playheadChange` event is broadcast by the MediaController and MediaPlayback components when the playback slider is moved by the user. (See `Media.playheadChange`.)

The MediaDisplay component works in conjunction with the MediaController component. Combined, the components behave in a manner similar to the MediaPlayback component, yet allow more flexibility with respect to layout. Therefore, if you require a flexible look and feel when presenting your media, use the MediaDisplay and MediaController components. Otherwise, the MediaPlayback component is the best choice.

Understanding the MediaDisplay component

When you place a MediaDisplay component on the Stage, it is drawn with no visible user interface. It is simply a container to hold and play media. The appearance of any video media playing in a MediaDisplay component is affected by the following properties:

- `Media.aspectRatio`
- `Media.autoSize`
- Height
- Width

Note: The user will not be able to 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 will always readjust the size of the playing media after the component size has been set to ensure that the aspect ratio of the media is maintained.

For FLV files, when `Media.autoSize` is set to `true`, the media to be played will be displayed 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 size. When `Media.autoSize` is set to `false`, the instance size will be used 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 will be 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 to 100, with 0 being mute and 100 being the 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 is hovering 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. The expanded state 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
- Playhead location indicator
- A slider, which users can drag to navigate around the media

The following items are also provided with the MediaController component:

- A Play/Pause state button
- A group of two buttons: Go to Beginning and Go to End, which navigate to the beginning and end of the media, respectively
- A volume control that consists of a slider, a mute, 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 movie background to display from behind the controls.

The MediaController component has an orientation setting, `Media.horizontal`, which can be used 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 the media from bottom to top.

The MediaDisplay and MediaController components can be associated with each other through the `Media.associateDisplay()` and `Media.associateController()` methods. When called, 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 the setting made by the user from the MediaController.

Understanding the MediaPlayback component

The MediaPlayback component is a combination of the MediaController and MediaDisplay controls. Both subcomponents are contained within MediaPlayback. The MediaController and MediaDisplay portions always scale to fit the size of the overall MediaPlayback component instance.

The MediaPlayback component uses `Media.controlPlacement` to determine the layout of the controls. Possible control placement include `top`, `bottom`, `left`, and `right`, indicating where the controls will be drawn in relation to the display. For example, a value of `right` would give a control a vertical orientation and position 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, there is generally a desire to provide users a method to stream the media and then control it. The following are example usage scenarios for media components:

- 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 for your clients that allows website users to preview DVDs and CDs that you sell in a rich media environment. The example below shows the steps to accomplish this and assumes your website is ready for inserting streaming components.

To create a Flash document that displays a CD or DVD preview:

1. In Flash, select File > New; then select Flash Document.
2. Open the Components panel (Window > Development Panels > Components) and double-click the MediaPlayback component, which places an instance of the component on the Stage.
3. Select the MediaPlayback component instance and enter the instance name **myMedia** in the Property inspector.
4. In the Component Inspector panel (Window > Development Panels > 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 that will be used in conjunction with a listener to open a pop-up window that informs the user that the movie is on sale. Give the cue point the name **cuePointName** and set the cue point time such that it is toward the end of the clip (within a few seconds). To accomplish this, take the following steps:
   a. Double-click a Window component to make it appear on the Stage.
   b. Delete the Window component. This places an item called Window in your library.
   c. Create a text box and write some text informing the user that the movie is on sale.
   d. Convert this text box to a movie clip by selecting Modify > Convert to Symbol, and give it the name **mySale_mc**.
   e. Right-click the `mySale_mc` movie clip in the library, select Linkage, and select the Export for ActionScript option. This places the movie clip in your runtime library.

10 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 fire off 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

Suppose you decide that you want more control over the look and feel of your media display. For this reason, you need to use the MediaDisplay and MediaController together to provide the desired experience. The following example shows the equivalent steps from the previous example that will create 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 (Window > Development Panels > Components), double-click the MediaController and MediaDisplay components, which places an instance of each component 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 Launch the Component Inspector panel 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 in 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 panel, set your orientation to vertical by setting the `horizontal` property to `false`.

10 In the Component Inspector panel, set `backgroundStyle` to None.

This specifies that the MediaController instance should not draw a background but should instead fill the media between the controls.

11 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:

a Select the MediaDisplay instance and, in the Property inspector, enter the instance name **myMediaDisplay**.

b Select the MediaController instance that will trigger the behavior.

c In the Behaviors panel (Window > Development Panels > Behaviors), click the Add (+) button and select Media > Associate Display.

d 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 333.

Using the Component Inspector panel with media components

The Component Inspector panel 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 panel 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 panel:

- The video time control is removed when an MP3 video type is selected, because this information is automatically read in when MP3 files are used. For FLV files, you must input 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.

- 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 the Milliseconds option 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 only set the FPS property by using the Component Inspector panel. Setting an fps value by using ActionScript has no effect and will be ignored.

Controlling media components by using behaviors

Behaviors are prewritten ActionScript scripts that you add to an object instance, such as a MediaDisplay component, to control that object. Behaviors allow you to 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 (Window > Development Panels > Behaviors), 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 to the MediaController component, you do not need to associate the MediaController component to 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 (Window > Development Panels > Behaviors), 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 panel, 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 movie, enter 5 in the SS text box and enter **myLabeledFrame** in the Name text box.

5 In the Behaviors panel (Window > Development Panels > Behaviors), 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 panel, 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 movie, enter 5 in the SS text box and enter **mySlide** in the Name text box.

6 In the Behaviors panel (Window > Development Panels > Behaviors), 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 component 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 component 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.
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 based on 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 component 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.
Media.controllerPolicy	Boolean	true	Determines whether the controller opens or closes based on 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 fits 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 (Window > Development Panels > Components), double-click the MediaPlayback component to add it to the Stage.
3. In the Property inspector, enter the instance name **myMedia**.
4. In the Property inspector, click Launch Component Inspector.
5. In the Component Inspector panel, enter **http://www.cathphoto.com/c.flv** in the URL text box.
6. 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 3, "Customizing Components," on page 27.

Using styles with media components

Styles are not supported with media components.

Using skins with media components

The media components do not support dynamic skinning, although you can open the media components 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 will 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.fla
- Macintosh: HD Drive:Users:*Username*:Library:Application Support:Macromedia:Flash MX 2004:*language*:Configuration:ComponentFLA.fla

For more information on component skins, see "About skinning components" on page 36.

Media class (Flash Professional only)

Inheritance mx.core.UIComponent

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 the version property, use the following code:

```
trace(mx.controls.MediaPlayback.version);
```

Note: The code `trace(myMediaInstance.version);` returns `undefined`.

Method summary for 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.play()`	MediaDisplay, MediaPlayback	Plays the media associated with the component instance at a given starting point.
`Media.pause()`	MediaDisplay, MediaPlayback	Pauses the playhead at its current location in the media Timeline.
`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

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 how the media-viewing portion of the MediaDisplay or MediaPlayback component sizes itself.
Media.backgroundStyle	MediaController	Determines if the component instance draws its chrome background.
Media.bytesLoaded	MediaDisplay, MediaPlayback	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 controls within the component are hidden during playback and only shown when a mouse-over event is triggered, or whether the controls are visible or hidden at all times.
Media.controlPlacement	MediaPlayback	Determines where the controls for the component are positioned in relation to the component.
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	Returns a Boolean value to indicate whether a given component instance is playing media.
Media.preferredHeight	MediaDisplay, MediaPlayback	The default value of the height of a FLV media file.
Media.preferredWidth	MediaDisplay, MediaPlayback	The default value of the width of a FLV media file.

Property	Components	Description
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

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	Is 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 Boolean value that determines what state the MediaController component is in when it is loaded at runtime. A `true` value indicates that the MediaController component should be in a play state at runtime, and a `false` value indicates that it is in a paused state at runtime. This property should be set in conjunction with the `autoPlay` property, such that both are either paused or playing at runtime. The default value is `true`.

Example

The following example indicates that the control will be paused when first loaded at runtime:

```
myMedia.activePlayControl = false;
```

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 can be used to name the cue point.

cuePointTime A number, expressed in seconds, which indicates when a `cuePoint` event is broadcast.

Returns

Nothing.

Description

Method; adds a cue point object to a MediaPlayback or MediaDisplay component 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` at time = `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 indicates the instance name of the MediaController component to associate.

Returns

Nothing.

Description

Method; associates a MediaDisplay component instance with a given MediaController instance.

If you have associated 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 indicates the instance name of the MediaDisplay component to associate.

Returns

Nothing.

Description

Method; associates a MediaController component instance with a given MediaDisplay instance.

If you have associated 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 will immediately start attempting to buffer and play. A `true` value indicates that the control will buffer and play at runtime; a `false` value indicates the control will be stopped at runtime. This property depends on the `contentPath` and `mediaType` properties. If `contentPath` and `mediaType` are not set, no playback will occur at runtime. The default value is `true`.

Example

The following example indicates that the control will not be started when first loaded at runtime:

```
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 how the media-viewing portion of the MediaDisplay or MediaPlayback component sizes itself.

For the MediaDisplay component, the property behaves as follows:

- If you set this property to `true`, Flash will display 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 size.
- If you set this property to `false`, Flash will use 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 will be used.

For the MediaPlayback component, the property behaves as follows:

- If you set this property to `true`, Flash will display the media at its preferred size unless the playback media area is smaller than the preferred size. If this is the case, Flash will shrink 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 will go unused.
- If you set this property to `false`, Flash will use 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 will be filled. This area is defined as the area above the controls (in the default layout), with an 8-pixel margin around it that makes up the edges of the component.

The default value is `true`.

Example

The following example indicates that the control will not be 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 value of `"default"` indicates that the chrome background will be drawn for the MediaController instance, while a value of `"none"` indicates that no chrome background will be drawn. The default value is `"default"`.

This is not a style property and therefore will not be 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 in the `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

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. See the example below.

When the event is triggered, it automatically passes an event object (*eventObject*) to the handler. Each event object has a set of 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 about event objects, see "Event Objects" on page 562.

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

```
var myMediaListener = new Object()
myMediaListener.click = function(){
  // insert your code here
}
myPlayer.addEventListener("click", myMediaListener);
```

Description

Event; broadcast when the user clicks the Play/Pause button. The detail field should 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 component 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 component 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

MediaController

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. The `Media.setMedia()` method is the only supported way of setting this property through ActionScript. The default value is `undefined`.

Example

The following example displays the name of the movie 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 displays itself when the user moves the mouse over the controller's collapsed state.

The possible values for this property are as follows:

- `"on"` indicates that the controls are always expanded.
- `"off"` indicates that the controls are always collapsed.
- `"auto"` indicates that the control will remain in the collapsed state until the user mouses 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 will keep 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 will be 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 cue point object.

`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[N]
```

Description

Property; an array of cue point objects that have been assigned to a MediaPlayback or MediaDisplay component instance. Within 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 component instance to full-screen mode. In other words, 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

None.

Returns

`cuePointName` The string that was provided when `Media.addCuePoint()` was used.

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 will display itself in a vertical or horizontal orientation. A `true` value indicates that the component will be displayed in a horizontal orientation; a `false` value indicates a vertical orientation. When set to `false`, the playhead and load progress indicator move from bottom to top. The default value is `true`.

Example

The following example will display 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; holds the value of the type of media to be played. The two choices are the FLV and MP3 formats. The default value is `"FLV"`. See “Importing Macromedia Flash Video (FLV) files” in Using Flash Help.

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 value 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 set to 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

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
myMedia.playing
```

Description

Read-only property; returns a Boolean value that indicates whether the media is playing. A value of `true` indicates that the media is playing; `false` indicates that the media is paused by the user.

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'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 parameters (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 instance 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'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 component 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;
}
```

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 specific 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 path and filename of the media to be played.

mediaType A string used to set the media type to either FLV or MP3. This parameter is optional.

Returns

Nothing.

Description

Method; sets the media type and path to the specified media type using a URL argument. The default value for *contentPath* is undefined.

This method provides the only supported way of setting the content path and media type for the MediaPlayback and MediaDisplay components.

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 time = 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, it is necessary to input `Media.totalTime` manually so that the playback slider 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`.

This property cannot be set 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 the volume setting integer value, which can range from 0 to 100. The default value for this property is 75.

Example

The following example sets the maximum volume for the 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 value between 0 and 100 that represents the volume level.

`type` The string `"volume"`.

Example

The following example will inform 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 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 366.

You cannot make the Menu component accessible to screen readers.

Interacting with the Menu component (Flash Professional only)

You can use the mouse and the keyboard to interact with a Menu component.

After a Menu 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 of 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 loops 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, or clicks and releases on a row if a submenu does not exist.

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 menus of individually selectable choices 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 presses a hotspot or 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 is composed of a data model and a view that displays the data. The Menu class is the view and contains the visual configuration methods. The MenuDataProvider class adds methods to the global XML prototype object (much like the DataProvider class 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 388.)

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>
```

Note: 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. This attribute is not available for the `check`, `radio`, or `separator` types.
`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.

Attribute name	Type	Default	Description
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 only required for the type radio.
selected	false, true, or false or true (a String or Boolean value)	false	A Boolean value indicating whether a check or radio item is on (true) or off (false). This attribute is not required.
enabled	false, true, or false or true (a String or Boolean value)	true	A Boolean value indicating whether this menu item can be selected (true) or not (false). This attribute is not required.

About menu item types

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 has nested subitems.

Separator menu items

Menu items whose type attribute is set to separator act as visual dividers 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

Menu items whose `type` attribute is set to `check` act as check box items within the menu; when the `selected` attribute is set to `true`, a checkmark 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 using the `setMenuItemSelected(item, b)` 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. A radio group is created 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 using 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 using the `setMenuItemSelected(item, b)` 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 object 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` must be unique within the scope of the root menu component instance (including all of the submenus of root).

About initialization object properties

The *`initObject`* (initialization object) parameter is a fundamental concept in creating the layout for the Menu component. The *`initObject`* 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 367.)

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"})
```

Is it important to note that 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 component parameters

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 374.

Creating an application with the Menu component

In the following example an application developer is building an application and uses the Menu component to expose some of the commands that users can issue, such as Open, Close, Save, and so on.

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 using ActionScript.
3. Drag a Button component from the Components panel to the Stage.

   Clicking button activates 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:

```
var listener = new Object();
listener.click = function(evtObj) {
    var button = evtObj.target;
  if(button.menu == undefined) {
        // Create a Menu instance and add some items
        button.menu = mx.controls.Menu.createMenu();
        button.menu.addMenuItem("Open");
        button.menu.addMenuItem("Close");
        button.menu.addMenuItem("Save");
        button.menu.addMenuItem("Revert");
        // Add a change-listener to catch item selections
        var changeListener = new Object();
        changeListener.change = function(event) {
      var item = event.menuItem;
      trace("Item selected:  " + item.attributes.label);
        }
        button.menu.addEventListener("change", changeListener);
    }
  button.menu.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. Select menu items to see the `trace` actions reporting which item was selected in the Output window.

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 using 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 using 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 an 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 firstChild
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 using 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

The menu sizes itself to fit 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 pixels 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. A Menu component supports the following Halo styles:

Style	Description
`themeColor`	The menu background color. This is the only color style that doesn't inherit its value.
`color`	The color of the text label of a menu item.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style; either `normal`, or `italic`.

Style	Description
fontWeight	The font weight; either normal, or bold.
rollOverColor	The rollover color of menu items.
selectionColor	Selected items and items that contain submenus.
selectionDisabledColor	Selected items and items that contain submenus and are disabled.
textRollOverColor	The color of text when you roll over an item.
textDecoration	The text decoration; either none, or underline.
textDisabledColor	The color of disabled text.
textSelectedColor	The color of text os a selected menu item.
popupDuration	The duration of the transition as a menu opens. The value 0 specifies no transition.

Using skins with the Menu component

For the latest information about this feature, click the Update button at the top of the Help tab.

Menu class (Flash Professional only)

Inheritance UIObject > UIComponent > View > ScrollView > ScrollSelectList > Menu

ActionScript Class Name mx.controls.Menu

Method summary for 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.
Menu.indexOf()	Returns the index of a given menu item.
Menu.removeAll()	Removes all items from a menu.
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.

Inherits all methods from UIObject, UIComponent, ScrollView, and ScrollSelectList.

Property summary for the Menu class

Property	Description
Menu.dataProvider	The data source for a menu.

Inherits all properties from UIObject, UIComponent, ScrollView, and ScrollSelectList.

Event summary for the Menu class

Event	Description
Menu.change	Broadcast when a user selects an item.
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.

Inherits all events from UIObject, UIComponent, ScrollView, and ScrollSelectList

Menu.addMenuItem()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenu.addMenu(initObject)
```

Usage 2:

```
myMenu.addMenu(childMenuItem)
```

Parameters

initObject An object containing properties that initialize a menu item's attributes. See "About menu item XML attributes" on page 367.

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(mySecondMenu, 3));
```

Menu.addMenuItemAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenu.addMenuItemAt(index, initObject)
```

Usage 2:

```
myMenu.addMenuItemAt(index, childMenuItem)
```

Parameters

index An integer indicating the order (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 367.

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 fourth child of the root of another menu:

```
myMenu.addMenuItemAt(3, mySecondMenu.getMenuItemAt(mySecondMenu, 3));
```

Menu.change

Availability

Flash Player 6 version 79.

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.

V2 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 a set of 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 Menu 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 about event objects, see "Event Objects" on page 562.

Example

In the following example, a handler called `listener` is defined and passed to the `myMenu.addEventListener()` method as the second parameter. The event object is captured by the `change` handler in the `event` parameter. When the `change` event is broadcast, a `trace` statement is sent to the Output panel, as follows:

```
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 version 79.

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* argument is omitted or null, the Menu is attached to the `_root` Timeline.

If the *mdp* argument is omitted or null, the menu does not have any items; you must call the `addMenu()` or `setDataProvider()` methods to populate the menu.

Example

In the following example, line 1 creates a MenuDataProvider which is an XML object decorated with the methods of the MenuDataProvider 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 clock of code creates an empty menu attached to `myParentClip`, fills it with the data source `myMDP`, and opens it at the coordinates 100, 20, as follows:

```
var myMDP = new XML();

var newItem = 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.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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.dataProvider
```

Description

Property; the data source for items in a Menu component.

The `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 API 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 version 79.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.hide()
```

Parameters

index The index of the Menu item.

Returns

Nothing.

Description

Method; closes a menu with optional transition effects.

Example

The following example retracts an extended menu:

```
myMenu.hide();
```

See also

`Menu.show()`

Menu.indexOf()

Availability

Flash Player 6 version 79.

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 version 79.

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.

V2 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 a set of 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, this value is undefined.
- `menu` A reference to the Menu instance that is hidden.

For more information about event objects, see "Event Objects" on page 562.

Example

In the following example, a handler called `form` is defined and passed to the `myMenu.addEventListener()` method as the second parameter. The event object is captured by `menuHide` handler in the `event` parameter. When the `menuHide` event is broadcast, a `trace` statement is sent to the Output panel, as follows:

```
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 version 79.

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.

V2 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 a set of 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 Menu, this value is undefined.
- `menu` A reference to the Menu instance that is shown.

For more information about event objects, see "Event Objects" on page 562.

Example

In the following example, a handler called `form` is defined and passed to the `myMenu.addEventListener()` method as the second parameter. The event object is captured by `menuShow` handler in the `evtObject` parameter. When the `menuShow` event is broadcast, a `trace` statement is sent to the Output panel, as follows:

```
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 version 79.

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.removeMenuItemAt()

Availability

Flash Player 6 version 79.

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 no item exists 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 version 79.

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.

V2 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 a set of 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` A reference to the menu item (XML node) that the pointer rolled off.

For more information about event objects, see "Event Objects" on page 562.

Example

In the following example, a handler called `form` is defined and passed to the `myMenu.addEventListener()` method as the second parameter. The event object is captured by the `rollOut` handler in the `event` parameter. When the `rollOut` event is broadcast, a `trace` statement is sent to the Output panel, as follows:

```
form = new Object();
form.rollOut = function(evt){
    trace("Menu rollOut: "+evt.menuItem.attributes.label);
}
myMenu.addEventListener("rollOut", form);
```

Menu.rollOver

Availability

Flash Player 6 version 79.

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.

V2 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 a set of 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` A reference to the menu item (XML node) that the pointer rolled over.

For more information about event objects, see "Event Objects" on page 562.

Example

In the following example, a handler called `form` is defined and passed to the `myMenu.addEventListener()` method as the second parameter. The event object is captured by the `rollOver` handler in the `event` parameter. When the `rollOver` event is broadcast, a `trace` statement is sent to the Output panel, as follows:

```
form = new Object();
form.rollOver = function(evt){
    trace("Menu rollOver: "+evt.menuItem.attributes.label);
}
myMenu.addEventListener("rollOver", form);
```

Menu.setMenuItemEnabled()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.setMenuItemEnabled(item, enable)
```

Parameters

item An XML node. The target menu item's node within the data provider.

enable A Boolean value indicating whether item is enabled (`true`) or not (`false`).

Returns

Nothing.

Description

Method; changes the target item's `enabled` attribute to the state given by 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.setMenuItemSelected(item, select)
```

Parameters

item An XML node. The target menu item's node within the data provider.

select A Boolean value indicating whether item is selected (`true`) or not (`false`). If the item is a check box, its check box is visible or not visible. If the item is a radio button, the item 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 version 79.

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 given location within 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 extends a menu:

```
myMenu.show(10, 10);
```

See also

`Menu.hide()`

MenuDataProvider class

The MenuDataProvider class is a decorator (mix-in) API that adds functionality to the XMLNode global class. This functionality lets XML instances assigned to a Menu dataProvider property manipulate their own data as well as the associated Menu views through the MenuDataProvider API.

Key concepts:

- The MenuDataProvider is a decorator (mix-in) API. It does not need to be instantiated to be used.
- Menus natively accept XML as a `dataProvider` property.
- If a Menu class is instantiated, all XML instances in the SWF file are decorated by the MenuDataProvider API.
- Only MenuDataProvider API methods broadcast events to the Menu controls. You can still use Native XML methods, but they are not broadcast events that refresh the Menu views.
  - Use MenuDataProvider API methods to control the data model.
  - Use XML methods for read-only operations like moving through the Menu hierarchy.
- All items in the Menu are XML objects decorated with the MenuDataProvider API.
- Changes to item attributes are not be reflected in the onscreen menu until a repaint occurs.

Method summary for the MenuDataProvider class

Method	Description
`MenuDataProvider.addMenuItem()`	Adds a child item.
`MenuDataProvider.addMenuItemAt()`	Adds a child item at a specific 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 version 79.

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenu.addMenuItem(initObject)
```

Usage 2:

```
myMenu.addMenuItem(childMenuItem)
```

Parameters

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 367.

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.

Example

The following example adds a new node to a specified node in the menu:

```
myMenuDP.firstChild.addMenuItem("Inbox", { label:"Item 1",
  icon:"radioItemIcon", type:"radio", selected:false, enabled:true,
  instanceName:"radioItem1", groupName:"myRadioGroup" } );
```

MenuDataProvider.addMenuItemAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myMenu.addMenuItemAt(index, initObject)
```

Usage 2:

```
myMenu.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 367.

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.

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", type:"radio", selected:false,
  enabled:true, instanceName:"radioItem1", groupName:"myRadioGroup" } );
```

MenuDataProvider.getMenuItemAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.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.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.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 within this parent menu item.

Example

The following example adds a menu item to a parent item and gets the item's index:

```
var myItem = myParentItem.addMenuItem({label:"That item"});
var myIndex = myParentItem.indexOf(myItem);
```

MenuDataProvider.removeMenuItem()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.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.

Example

The following example removes `myMenuItem` from its parent:

```
myMenuItem.removeMenuItem();
```

MenuDataProvider.removeMenuItemAt()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenu.removeMenuItemAt(index)
```

Parameters

index The index of the Menu item.

Returns

A reference to the removed menu item. This value is undefined if no item exists 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.

Example

The following example removes the fourth item:

```
myMenuDP.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 File and Edit menu bars in most common software applications (such as Macromedia Flash). The menu bar 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 menu bar 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 within 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 an menu activator is pressed, the corresponding menu opens below it. The menu stays active until the activator is pressed 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 menu bar 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. Both mouse and keyboard interactivity work together to let the user jump from menu to menu within the MenuBar component.

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 the 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, and so on) to the top edge of an application.

MenuBar component parameters

The following are authoring parameters that you can set for each MenuBar component instance in the Property inspector or in the Component Inspector panel:

labels An array that adds menu activators to the MenuBar with the given labels. The default value is [] (empty array).

You can write ActionScript to control these and additional options for the MenuBar component using its properties, methods, and events. For more information, see "MenuBar class" on page 395.

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 to 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, 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 menu bar instance. It then uses the Menu API 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 based on 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 a fixed size that is dependent on the font styles and the text length.

Using styles with the MenuBar component

The MenuBar creates an activator label for each menu within a group. You can use styles to change the look of the activator labels. A MenuBar component supports the following Halo styles:

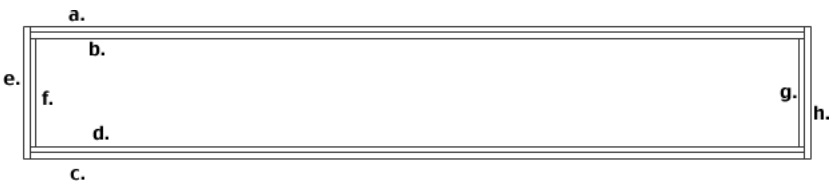
Style	Description
`themeColor`	The selection highlight color. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style; either "normal", or "italic".

Style	Description
fontWeight	The font weight; either "normal", or "bold".
textDecoration	The text decoration; either "none", or "underline".
popupDuration	The amount of time in milliseconds that it takes a menu to pop up. The default value is 0.

The MenuBar component also uses the RectBorder class to draw inset and outset highlights around the label when a user interacts with it. You can use the `setStyle()` method (see `UIObject.setStyle()`) to change the following RectBorder style properties:

RectBorder styles	Border position
borderColor	a
highlightColor	b
borderColor	c
shadowColor	d
borderCapColor	e
shadowCapColor	f
shadowCapColor	g
borderCapColor	h

The style properties set the following positions on the border:



Using skins with the MenuBar component

The MenuBar component uses the skins of the Menu component to represent its visual states. For information about the Menu component skins, see "Using skins with the Menu component" on page 374.

MenuBar class

Inheritance UIObject > UIComponent > MenuBar

ActionScript Class Name mx.controls.MenuBar

Method summary for the MenuBar class

Method	Description
MenuBar.addMenu()	Adds a menu to the menu bar.
MenuBar.addMenuAt()	Adds a menu to the menu bar at a specific location.
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 from a menu bar at a specified location.
MenuBar.setMenuEnabledAt()	A Boolean value indicating whether a menu is enabled (true) or not (false).

Inherits all methods from UIObject and UIComponent.

Property summary for 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 bar item.
MenuBar.labelFunction	A function that determines what to display as the label of each menu bar item.

Inherits all methods from UIObject and UIComponent.

MenuBar.addMenu()

Availability

Flash Player 6 version 79.

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 firstChild 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 with the value specified in the *label* parameter. 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 the `Menu.addMenuItem()` method to add the menu items New and Open:

```
menu = myMenuBar.addMenu("File");
menu.addMenuItem({label:"New", instanceName:newInstance"});
menu.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 `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 version 79.

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 `firstChild` 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 value specified in the *label* parameter. 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* `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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.dataProvider
```

Description

Property; the data model for items in a MenuBar component.

The `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 API 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){
    myMenuBarDP.dataProvider = myMenuBarDP;
  } else {
  trace("error loading XML file");
  }
}
```

MenuBar.getMenuAt()

Availability

Flash Player 6 version 79.

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 the `getMenuAt()` method returns an instance, it is possible to add items to a menu at the specified index. In the following example, after using the Label 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.getMenuEnabledAt(index)
```

Parameters

index The index of the MenuBar item.

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 version 79.

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 version 79.

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 builds 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.removeMenuAt(index)
```

Parameters

index The index of the MenuBar item.

Returns

A reference to the returned MenuBar item. This value is undefined if no item exists in that position.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myMenuBar.setMenuEnabledAt(index, boolean)
```

Parameters

index The index of the MenuBar item to set.

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 given index. If there is no menu at that index, calling this method has no effect.

Example

The following example gets the MenuBarColumn object at index 3:

```
myMenuBar.setMenuEnabledAt(3);
```

NumericStepper component

The NumericStepper component allows a user to step through an ordered set of numbers. The component consists of a number displayed beside small up and down arrow buttons. When a user pushes the buttons, the number is raised or lowered incrementally. If the user clicks either of the arrow buttons, the number increases or decreases, based on the value of the stepSize parameter, until the user releases the mouse or until the maximum or minimum value is reached.

The NumericStepper only handles 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	Value changes by one unit.
Left	Moves the insertion point to the left within the text box.
Right	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	Value changes by one unit.

For more information about controlling focus, see "Creating custom focus navigation" on page 24 or "FocusManager class" on page 270.

A live preview of each stepper instance reflects the value of the value parameter indicated by the Property inspector or Component Inspector panel while authoring. However, there is no mouse or keyboard interaction with the stepper 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 only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

Using the NumericStepper component

The NumericStepper can be used 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 their credit card expiration date. In another example, you could use a NumericStepper to allow a user to increase or decrease a font size.

NumericStepper parameters

The following are authoring parameters that you can set for each NumericStepper component instance in the Property inspector or in the Component Inspector panel:

value sets the value of the current step. The default value is 0.

minimum sets the minimum value of the step. The default value is 0.

maximum sets the maximum value of the step. The default value is 10.

stepSize sets the unit of change for the step. The default value is 1.

You can write ActionScript to control these and additional options for NumericStepper components using its properties, methods, and events. For more information, see NumericStepper class.

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 Button component, do the following:

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 both 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.

Resizing the NumericStepper component does not change the size of the down and up arrow buttons. If the stepper is resized greater than the default height, the stepper buttons are pinned to the top and the bottom of the component. The stepper 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 stepper instance. If the name of a style property ends in "Color", it is a color style property and behaves differently than non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

A NumericStepper component supports the following Halo styles:

Style	Description
`themeColor`	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style; either "normal", or "italic".
`fontWeight`	The font weight; either "normal", or "bold".
`textDecoration`	The text decoration; either "none", or "underline".
`textAlign`	The text alignment; either "left", "right", or "center".

Using skins with the NumericStepper component

The NumericStepper component skins to represent its visual states. To skin the NumericStepper component while authoring, modify skin symbols in the library and re-export the component as a SWC. The skin symbols are located in the Flash UI Components 2/Themes/MMDefault/Stepper Elements/states folder in the library. For more information, see "About skinning components" on page 36.

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 clicked. 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 uses the following skin properties:

Property	Description
`upArrowUp`	The up arrow's up state. The default value is `StepUpArrowUp`.
`upArrowDown`	The up arrow's pressed state. The default value is `StepUpArrowDown`.
`upArrowOver`	The up arrow's over state. The default value is `StepUpArrowOver`.
`upArrowDisabled`	The up arrow's disabled state. The default value is `StepUpArrowDisabled`.
`downArrowUp`	The down arrow's up state. The default value is `StepDownArrowUp`.
`downArrowDown`	The down arrow's down state. The default value is `StepDownArrowDown`.

Property	Description
downArrowOver	The down arrow's over state. The default value is StepDownArrowOver.
downArrowDisabled	The down arrow's disabled state. The default value is StepDownArrowDisabled.

NumericStepper class

Inheritance UIObject > UIComponent > NumericStepper

ActionScript Class Name mx.controls.NumericStepper

The properties of the NumericStepper class allow you to add indicate the minimum and maximum step values, the unit amount for each step, and the current value of the step at runtime.

Setting a property of the NumericStepper class with ActionScript overrides the parameter of the same name set in the Property inspector or Component Inspector panel.

The NumericStepper component uses the FocusManager 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 24.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.NumericStepper.version);
```

Note: The following code returns undefined: `trace(myNumericStepperInstance.version);`.

Method summary for the NumericStepper class

Inherits all methods from UIObject and UIComponent.

Property summary for 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 step.
NumericStepper.value	A number indicating the current value of the stepper.

Inherits all properties from UIObject and UIComponent.

Event summary for the NumericStepper class

Event	Description
`NumericStepper.change`	Triggered when the value of the step changes.

Inherits all events from UIObject and UIComponent.

NumericStepper.change

Availability

Flash Player 6 version 79.

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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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` action 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 the `UIEventDispatcher.addEventListener()` method from `myNumericStepper` and passes it the `change` event and the `form` listener object as parameters, as in the following:

```
form = new Object();
form.change = function(eventObj){
  // eventObj.target is the component which 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 version 79.

Edition

Flash MX 2004.

Usage

stepperInstance.maximum

Description

Property; the maximum range value of the stepper. This property can contain a number with 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 version 79.

Edition

Flash MX 2004.

Usage

stepperInstance.minimum

Description

Property; the minimum range value of the stepper. This property can contain a number with 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 version 79.

Edition

Flash MX 2004.

Usage

stepperInstance.nextValue

Description

Property (read-only); the next sequential value. This property can contain a number with 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 version 79.

Edition

Flash MX 2004.

Usage

stepperInstance.previousValue

Description

Property (read-only); the previous sequential value. This property can contain a number with 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 version 79.

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 with up to three decimal places.

Example

The following example sets the current `value` 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 version 79.

Edition

Flash MX 2004.

Usage

stepperInstance.value

Description

Property; the current value displayed in the text area of the stepper. The value will not be 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 with 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 allows you to create overlapping windows that can be modal or non-modal. (A modal window doesn't allow interaction with other windows while it's active.) You can call `PopUpManager.createPopUp()` to create an overlapping window, and call `PopUpManager.deletePopUp()` on the window instance to destroy a pop-up window.

Method summary for the PopUpManager class

Event	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 version 79.

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 non-modal, 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 version 79.

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 and 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 loading progress while a user waits for the content to load. The loading process can be determinate or indeterminate. A determinate progress bar is a linear representation of the progress of a task 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.

Components are set to export in first frame by default. 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 will need to deselect Export in first frame in each component's Linkage Properties dialog (Library panel options > Linkage). The ProgressBar, however, should be set to Export in first frame, because it must display first while other content streams into Flash Player.

A live preview of each ProgressBar instance reflects changes made to parameters in the Property inspector or Component Inspector panel while authoring. The following parameters are reflected in the live preview: conversion, direction, label, labelPlacement, mode, and source.

Using the ProgressBar component

A progress bar allows you to 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" and "polled". 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

The following are authoring parameters that you can set for each ProgressBar component instance in the Property inspector or in the Component Inspector panel:

mode 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 A string to be converted into an object representing the instance name of the source.

direction The direction toward which the progress bar fills. This value can be right or left; the default value is right.

label The text indicating the loading progress. This parameter is a string in the format "%1 out of %2 loaded (%3%%)"; %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 "??". If a value is undefined, the label doesn't display.

labelPlacement 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 A number 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 ProgressBar components using its properties, methods, and events. For more information, see ProgressBar class.

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, 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. The Loader component emits these events. For more information, see "Loader component" on page 314.

To create an application with the ProgressBar component in event mode, do the following:

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 that 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 the load method 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, do the following:

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 that creates a Sound object called `loader` and calls the `loadSound()` method 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, do the following:

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 that updates the progress bar manually on every file download using calls to the `setProgress()` method:

```
for(var:Number i=1; i <= total; i++){
  // insert code to load file
  // insert code to load file
pBar.setProgress(i, total);
}
```

Customizing the ProgressBar component

You can transform a ProgressBar component horizontally both 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 left cap and right cap of the progress bar and track graphic are a fixed size. When you resize a progress bar, the middle part of the progress bar resized to fit between them. 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 non-color style properties. For more information, see “Using styles to customize component color and text” on page 27.

A ProgressBar component supports the following Halo styles:

Style	Description
themeColor	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
color	The text of a component label.
disabledColor	The disabled color for text.
fontFamily	The font name for text.
fontSize	The point size for the font.
fontStyle	The font style; either "normal" or "italic".
fontWeight	The font weight; either "normal" or "bold".
textDecoration	The text decoration; either "none" or "underline".

Using skins with the ProgressBar component

The ProgressBar component uses the following movie clip symbols to display its states: TrackMiddle, TrackLeftCap, TrackRightCap and BarMiddle, BarLeftCap, BarRightCap and IndBar. The IndBar symbol is used for an indeterminate progress bar. To skin the ProgressBar component while authoring, modify symbols in the library and re-export the component as a SWC. The symbols are located in the Flash UI Components 2/Themes/MMDefault/ProgressBar Elements folder in the library of the HaloTheme.fla file or the SampleTheme.fla file. For more information, see "About skinning components" on page 36.

If you use the `UIObject.createClassObject()` method to create a ProgressBar component instance dynamically (at runtime), you can also skin it dynamically. To skin a component at runtime, set the skin properties of the *`initObject`* parameter that is passed to the `createClassObject()` method. The skin properties set the names of the symbols to use as the states of the progress bar.

A ProgressBar component uses 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`.

ProgressBar class

Inheritance UIObject > 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 panel.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.ProgressBar.version);
```

Note: The following code returns undefined: `trace(myProgressBarInstance.version);`.

Method summary for the ProgressBar class

Method	Description
`ProgressBar.setProgress()`	Sets the progress of the bar in manual mode.

Inherits all methods from UIObject.

Property summary for 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 that the progress bar fills.
`ProgressBar.indeterminate`	Indicates that the total bytes of the source is unknown.
`ProgressBar.label`	The text the 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`	A number indicating the percent loaded.
`ProgressBar.source`	The content to load whose progress is monitored by the progress bar.
`ProgressBar.value`	Indicates the amount of progress that has been made. This property is read-only.

Inherits all properties from UIObject.

Event summary for the ProgressBar class

Event	Description
ProgressBar.complete	Triggered when loading is complete.
ProgressBar.progress	Triggered as content loads in event or polled mode.

Inherits all events from UIObject.

ProgressBar.complete

Availability

Flash Player 6 version 79.

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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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, as in the following:

```
form.complete = function(eventObj){
  // eventObj.target is the component which generated the change event,
  // i.e., the Progress Bar.
  trace("Value changed to " + eventObj.target.value);
}
pBar.addEventListener("complete", form);
```

See also

`UIEventDispatcher.addEventListener()`

ProgressBar.conversion

Availability

Flash Player 6 version 79.

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.

Example

The following code displays the value of the loading progress in kilobytes:

```
pBar.conversion = 1024;
```

ProgressBar.direction

Availability

Flash Player 6 version 79.

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 sets makes the progress bar fill from right to left:

```
pBar.direction = "left";
```

ProgressBar.indeterminate

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

pBarInstance.indeterminate

Description

Property; a Boolean value that indicates whether the progress bar has a candy-cane striped fill and a loading source of unknown size (true), or a solid fill and a loading source of a known size (false).

Example

The following code creates a determinate progress bar with a solid fill that moves from left to right:

```
pBar.direction = "right";
pBar.indeterminate = false;
```

ProgressBar.label

Availability

Flash Player 6 version 79.

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%%)"; %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 "??". If a value is undefined, the label doesn't display. The default value is `"LOADING %3%%"`

Example

The following code sets the text that appears beside the progress bar to the format "4 files loaded":

```
pBar.label = "%1 files loaded";
```

See also

ProgressBar.labelPlacement

ProgressBar.labelPlacement

Availability

Flash Player 6 version 79.

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 sets label to display 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 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

pBarInstance.minimum

Description

Property; the smallest progress 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 version 79.

Edition

Flash MX 2004.

Usage

pBarInstance.mode

Description

Property; the mode in which the progress bar loads content. This value can be one of the following: "event", "polled", or "manual". The most commonly used modes are "event" and "polled". These modes use the source parameter to specify a loading process that either emits progress and complete events, like a Loader component (event mode), or exposes getBytesLoaded and getsBytesTotal methods, like a MovieClip object (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.

A Loader object should be used as the source in event mode. Any object that exposes getBytesLoaded() and getBytesTotal() methods can be used as a source in polled mode. (Including a custom object or the _root object)

Example

The following code sets the progress bar to event mode:

```
pBar.mode = "event";
```

ProgressBar.percentComplete

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

pBarInstance.percentComplete

Description

Property (read-only); returns the percentage of completion of the process. This value is floored. The following is the formula 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 version 79.

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 only broadcast 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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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, as follows:

```
var form:Object = new Object();
form.progress = function(eventObj){
  // eventObj.target is the component which generated the change event,
  // i.e., the Progress Bar.
  trace("Value changed to " + eventObj.target.value);
}
pBar.addEventListener("progress", form);
```

See also

`UIEventDispatcher.addEventListener()`

ProgressBar.setProgress()

Availability

Flash Player 6 version 79.

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 percent.

Returns

A number indicating the amount of progress that has been made.

Description

Method; sets the state of the 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. The argument completed is assigned to value property and argument total is assigned to the maximum property. The minimum property is not altered.

Example

The following code calls the `setProgress()` method based on the progress of a Flash application's Timeline:

```
pBar.setProgress(_currentFrame, _totalFrames);
```

ProgressBar.source

Availability

Flash Player 6 version 79.

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 can be used with contents 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 version 79.

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 allows you to force a user to make a single choice within a set of choices. The RadioButton 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 can set the `groupName` parameter to indicate which group a radio button belongs to.

A radio button can be enabled or disabled. When a user tabs into a radio button group, only the selected radio button receives focus. A user can press the arrow keys to change focus within the group. In the disabled state, a radio button doesn't receive mouse or keyboard input.

A RadioButton component group receives focus if you click it or tab to it. When a RadioButton group has focus, you can use the following keys control it:

Key	Description
Up/Right	The selection moves to the previous radio button within the radio button group.
Down/Left	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 24 or "FocusManager class" on page 270.

A live preview of each RadioButton instance on the Stage reflects changes made to parameters in the Property inspector or Component Inspector panel while 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 427.

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 only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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 is using to pay.

RadioButton parameters

The following are authoring parameters that you can set for each RadioButton component instance in the Property inspector or in the Component Inspector panel:

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 within a group can have a selected value of true. If more than one radio button within 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.

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 a 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, do the following:

1. Drag two RadioButton components from the Components panel to the Stage.
2. Select one of the radio buttons and in the Component Inspector panel do the following:
   - Enter Yes for the label parameter.
   - Enter Flashist for the data parameter.
3. Select the other radio button and in the Component Inspector panel 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 the TextArea component instance `theVerdict` 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 both 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()" on page 576).

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 clips 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 non-color style properties. For more information, see "Using styles to customize component color and text" on page 27.

A RadioButton component uses the following Halo styles:l

Style	Description
`themeColor`	The background of a component. This is the only color style that doesn't inherit its value. Possible values are "haloGreen", "haloBlue", and "haloOrange".
`color`	The text of a component label.
`disabledColor`	The disabled color for text.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style; either "normal", or "italic".
`fontWeight`	The font weight; either "normal", or "bold".

Using skins with the RadioButton component

The RadioButton component can be skinned 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 36.

If a radio button is enabled and unselected, it displays its roll-over 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 within 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.

If you use the `UIObject.createClassObject()` method to create a RadioButton component instance dynamically, you can also skin the component dynamically. To skin a RadioButton component dynamically, pass skin properties to the `UIObject.createClassObject()` method. For more information, see "About skinning components" on page 36. The skin properties indicate which symbol to use to display a component.

A RadioButton component uses the following skin properties:

Name	Description
falseUpIcon	The unchecked state. The default value is radioButtonFalseUp.
falseDownIcon	The pressed-unchecked state. The default value is radioButtonFalseDown.
falseOverIcon	The over-unchecked state. The default value is radioButtonFalseOver.
falseDisabledIcon	The disabled-unchecked state. The default value is radioButtonFalseDisabled.
trueUpIcon	The checked state. The default value is radioButtonTrueUp.
trueDisabledIcon	The disabled-checked state. The default value is radioButtonTrueDisabled.

RadioButton class

Inheritance UIObject > UIComponent > SimpleButton > Button > 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 panel.

The RadioButton component uses the FocusManager 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 24.

Each component class has a `version` property which is a class property. Class properties are only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.RadioButton.version);
```

Note: The following code returns undefined: `trace(myRadioButtonInstance.version);`.

Method summary for the RadioButton class

Inherits all methods from UIObject, UIComponent, SimpleButton, and Button class.

Property summary for 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 or radio button instance.
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.

Property	Description
RadioButton.selected	Sets the state of the radio button instance to selected and deselects the previously selected radio button.
RadioButton.selectedData	Selects the radio button in a radio button group with the specified data value.
RadioButton.selection	A reference to the currently selected radio button in a radio button group.

Inherits all properties from UIObject, UIComponent, SimpleButton, and the Button class

Event summary for the RadioButton class

Event	Description
RadioButton.click	Triggered when the mouse is pressed over a button instance.

Inherits all events from UIObject, UIComponent, SimpleButton, and Button class

RadioButton.click

Availability

Flash Player 6 version 79.

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 using 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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example, written on a frame of the Timeline, sends a message to the Output panel when a radio button in the `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` action 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. You can access instance properties from the `target` property (in this example, the `RadioButton.selection` property is accessed) The last line calls the `UIEventDispatcher.addEventListener()` method from `radioGroup` and passes it the `click` event and the `form` listener object as parameters, as in the following:

```
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`, as in the following:

```
on(click){
  trace("radio button component was clicked");
}
```

RadioButton.data

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

radioButtonInstance.data

Description

Property; specifies the data to associate with a radio button instance. Setting this property overrides the data parameter value set while authoring in the Property inspector or in the Component Inspector panel. The `data` property can be 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 version 79.

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 a group name for a radio button group. Calling this method overrides the groupName parameter value set while 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 with 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 version 79.

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 while authoring. If the label text is too long to fit within the bounding box of the component, the text clips.

Example

The following example sets the label property of the instance `radioButton`:

```
radioButton.label = "Remove from list";
```

RadioButton.labelPlacement

Availability

Flash Player 6 version 79.

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 set 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 set 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 will clip.
- `"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 will clip.

Example

The following code places the label to the left of each radio button in the `radioGroup`:

```
radioGroup.labelPlacement = "left";
```

RadioButton.selected

Availability

Flash Player 6 version 79.

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, as follows:

```
mcButton.selected = true;
trace(mcButton.selected);
```

RadioButton.selectedData

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

```
radioButtonInstance.selection
radioButtonGroup.selection
```

Description

Property; behaves differently if 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)
```

RDBMSResolver component (Flash Professional only)

You use resolver components in combination with the DataSet component (part of the data management functionality in the Macromedia Flash data architecture). The resolver components enable you to convert changes made to the data within your application into a format that is appropriate for the external data source that you are updating. These components have no visual appearance at runtime.

If you use a DataSet component in your application it generates an optimized set of instructions (DeltaPacket) that describes the changes made to the data at runtime. This set of instructions is converted to the appropriate format (update packet) by the resolver components. When an update is sent to the server, the server sends a response (result packet) containing additional updates or errors that result from the update operation. The resolver components can convert this information back into a DeltaPacket that can be applied to the DataSet component to keep it in sync with the external data source. Resolver components enable you to keep your application and an external data source in sync without writing additional ActionScript code.

The RDBMSResolver component translates XML that can be received and parsed by a web service, a JavaBean, a servlet, or an ASP page. The XML contains the necessary information and formatting for updating any standard SQL relational database. A parallel resolver component, XUpdateResolver (see “XUpdateResolver component (Flash Professional only)” on page 632), exists for returning data to an XML-based server. For more information about DataSet components, see “DataSet component (Flash Professional only)” on page 193. For more information about connectors, see “WebServiceConnector (Flash Professional only)” on page 604 and “XMLConnector component (Flash Professional only)” on page 624. For more information about the Flash data architecture, see “Resolver components (Flash Professional only)” in Using Flash Help.

The RDBMSResolver component converts changes made to the data in your application into an XML packet that can be sent to an external data source.

Note: You can use the RDBMSResolver to send data updates to any external data source that can parse XML and generate SQL statements against a database; for example, an ASP page, a Java servlet, or a ColdFusion component.

The updates from the RDBMSResolver component are sent in the form of an XML update packet communicated to the database through a connector object. The resolver 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.

Using the RDBMSResolver component (Flash Professional only)

Use this 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.

For more information on working with the RDBMSResolver component, see "Resolver components (Flash Professional only)" in Using Flash Help.

RDBMSResolver component parameters

TableName String representing the table name in the XML for the database table to be updated. This should be the same value as the input value for the `Resolver.fieldInfo` item to be updated. If no updates to this field exist, this value should be blank, which is the default value.

UpdateMode Enumerator that determines the way key fields are identified during the generation of the XML update packet. The default value is `umUsingKey`. Possible values are as follows:

- `umUsingAll` Uses the old values of all of the fields modified 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` This is the default value for this property. 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 String representing a null field value. This is customizable to prevent it from being confused with an empty string ("") or another valid value. The default value is `{_NULL_}`.

FieldInfo Collection representing one or more key fields that uniquely identify the records. If your data source is a database table, then it 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 RDBMSResolver component contains a FieldInfo property for this purpose. This collection property lets you define an unlimited number of fields with properties that identify fields that require special handling. Each FieldInfo item in the collection contains three properties:

- FieldName Name of a field. This should map to a field in the DataSet component.
- OwnerName Optional value used to identify fields not "owned" by the same table defined in the resolver component's TableName parameter. If this value is the same value as the TableName parameter or is blank, usually the field is included in the XML update packet. If it is 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 the 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 using a join in the query. This field should be excluded from the update, so 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.

Property summary for the RDBMSResolver component

Property	Description
RDBMSResolver.deltaPacket	A copy of the DataSet component's DeltaPacket property.
RDBMSResolver.fieldInfo	An unlimited number of fields with properties that identify DataSet fields requiring special handling as either a key field or a nonupdatable field.
RDBMSResolver.nullValue	Indicator that a field's value is null.
RDBMSResolver.tableName	The table name put in the XML for the database table to be updated.
RDBMSResolver.updateMode	The value that determines how key fields are identified when the XML update packet is being generated.
RDBMSResolver.updatePacket	A copy of the connector updatePacket property containing the latest XML-formatted data for return from the connector to the DataSet component after the server has received this application's request to update.
RDBMSResolver.updateResults	A copy of the connector's Results property, which returns any XML-formatted errors or updates for the DataSet component.

Method summary for the RDBMSResolver component

Method	Description
RDBMSResolver.addFieldInfo()	Adds a new item to the fieldInfo collection, used for setting up a resolver component dynamically at runtime, rather than using the Component inspector in the authoring environment.

Event summary for the RDBMSResolver component

Event	Description
RDBMSResolver.beforeApplyUpdates	Defined in your application; called by the resolver 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 resolver component to reconcile the updates between the updatePacket property sent to the server and the updatePacket property returned from the server.
RDBMSResolver.reconcileUpdates	Defined in your application; called by the resolver component to reconcile the update received by the server and the pending update.

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 this information object describes.

ownerName String; provides the name of the table that owns this field. May be left blank ("") if it is the same as the resolver instance's tablename property.

isKey Boolean; indicates whether this field is a key field.

Returns

None.

Description

Method; adds a new item to the XML fieldInfo collection in the update Packet. Use this method if you must set up a resolver component dynamically at runtime, rather than using the Component inspector in the authoring environment.

Example

The following example creates a resolver component and provides the name of the table, 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 resolver event object should contain the following properties:

Property	Description
`target`	Object; resolver firing this event.
`type`	String; name of the event.
`updatePacket`	XML object; XML object about to be applied.

Returns

None.

Description

Property; property of type `deltaPacket` that receives a `deltaPacket` to be translated into an `xupdatePacket`, and outputs a `deltaPacket` from any server results placed into 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 `deltaPacket` again so it can be resent the next time the `deltaPacket` 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 modified before being added to the next `deltaPacket`.

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; property of type `deltaPacket` that receives a `deltaPacket` to be translated into an `updatePacket`, and outputs a `deltaPacket` from any server results placed into the `updateResults` property.

Messages in the `updateResults` property are treated as errors. This means that a delta with messages is added to the `deltaPacket` again so it can be resent the next time the `deltaPacket` 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 modified before being added to the next `deltaPacket`.

RDBMSResolver.fieldInfo

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.fieldInfo

Description

Property; property of type `Collection` specifies a collection of an unlimited number of fields with properties that identify DataSet fields that require special handling, either because the field is a key field or a nonupdatable field. Each `FieldInfo` item in the collection contains three properties:

Property	Description
`FieldName`	Name of the special-case field. This field name should map to a field name in the DataSet.
`OwnerName`	This optional property is the name of the owner of this field if this field is not "owned" by the same table defined in the component `TableName` parameter. If this is filled in with the same value as that parameter or left blank, usually the field is included in the XML update packet. If filled in differently, this field is excluded from the update packet.
`IsKey`	Boolean value set to `true` for all key fields for the table to be updated.

RDBMSResolver.nullValue

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.deltaPacket`

Description

Property; property of type String used to provide a null value for a field's value. This is customizable 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 updatePackets This resolver event object should contain the following properties:

Property	Description
target	Object; resolver firing this event.
type	String; name of the event.

Returns

None.

Description

Event; called by the resolver component to compare two packets after results have been received from the server and have been applied to the deltaPacket.

A single updateResults packet can contain both results of operations that were in the deltaPacket, and information about updates performed by other clients. When a new updatePacket is received, the operation results and database updates are split into two updatePackets and placed separately into the deltaPacket property. The reconcileResults event is fired just before the deltaPacket containing the operation results is sent using data binding.

Example

The following example reconciles two updatePackets 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

Flash MX Professional 2004.

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 resolver event object should contain the following properties:

Property	Description
target	Object; resolver firing this event.
type	String; name of the event.

Returns

None.

Description

Event; called by the resolver component when results have been received from the server after applying the updates from a deltaPacket. A single updateResults packet can contain both results of operations that were in the deltaPacket, and information about updates that were performed by other clients. When a new updatePacket is received, the operation results and database updates are split into two deltaPackets, which are placed separately into the deltaPacket property. The reconcileUpdates event is fired just before the deltaPacket 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.deltaPacket

Description

Property; property of type String used to represent the table name in the XML for the database table to be updated. This also determines which fields to send in the `updatePacket`. To make this determination, the resolver compoent compares the value of this property to the value provided for the `fieldInfo.ownerName` property. If a field has no entry in the `fieldInfo` collection property, the field is placed into the `updatePacket`. If a field has an entry in the `fieldInfo` collection property, and the `ownerName` property value is blank or identical to the resolver component's `tableName` property, the field is placed into the `updatePacket`. If a field has an entry in the `fieldInfo` collection property, and the `ownerName` property value is not blank and is different from the resolver component's `tableName` property, the field is not placed into the `updatePacket`.

RDBMSResolver.updateMode

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.deltaPacket`

Description

Property; property containing several values that determine how key fields are identified when the XML update packet is being generated. The following three strings are values for this property:

Value	Description
`umUsingAll`	Uses the old values of all of the fields modified 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 all 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`	This is the default value for this property. 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 will overwrite any other user's changes to the same data.

RDBMSResolver.updatePacket

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.deltaPacket

Description

Description

Property; property of type XML 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.deltaPacket

Description

Property; property of type deltaPacket 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 DataSet; 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.

Messages in the updateResults property are treated as errors. This means that a delta with messages is added to the deltaPacket again so it can be resent the next time the deltaPacket 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 modified before being added to the next deltaPacket.

Remote Procedure Call (RPC) Component API

ActionScript Class Name mx.data.components.RPC

The Remote Procedure Call (RPC) Component API is an interface (a set of methods, properties, and events) that can be implemented by a Flash MX 2004 v2 component. The RPC Component API defines an easy way to send parameters to, and receive results from, an external resource such as a web service.

Components that implement the RPC API include the WebServiceConnector and XMLConnector components. These components act as connectors between an external data source, such as a web service or XML file, and a UI component in your application.

An RPC component has the ability to call a single external function, pass in parameters, and receive results. The component can call that same function multiple times. To call multiple functions, you must use multiple components.

Property summary for the RPC Component class

Property	Description
`RPC.multipleSimultaneousAllowed`	Indicates whether multiple calls can take place at the same time.
`RPC.params`	Specifies data that will be sent to the server when the next `trigger()` operation is executed.
`RPC.results`	Identifies data that was received from the server as a result of the a `trigger()` operation.
`RPC.suppressInvalidCalls`	Indicates whether to suppress a call if parameters are invalid.

Method summary for the RPC Component class

Method	Description
`RPC.trigger()`	Initiates a remote procedure call.

Event summary for the RPC Component class

Event	Description
`RPC.result`	Broadcast when a Remote Procedure Call completes successfully.
`RPC.send`	Broadcast when the `trigger()` function is in process, after the parameter data has been gathered but before the data is validated and the Remote Call is initiated.
`RPC.status`	Broadcast when a Remote Procedure Call is initiated, to inform the user of the status of the operation.

RPC.multipleSimultaneousAllowed

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.multipleSimultaneousAllowed;
```

Description

Property; indicates whether multiple calls can take place at the same time. If false, then the `trigger()` function will perform a call if another call is already in progress. A `status` event will be emitted, with the code `CallAlreadyInProgress`. If true, then the call will take place.

When multiple calls are simultaneously in progress, there is no guarantee that they will complete in the same order as they were triggered. Also, Flash Player may place limits on the number of simultaneous network operations. This limit varies by version and platform.

RPC.params

Availability

Flash Player 7.

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.

RPC.result

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("result", myListenerObject);
```

Description

Event; broadcasts when an Remote Procedure Call operation successfully completes.

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.

RPC.results

Availability

Flash Player 7.

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 signalled 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 2 ways:

- Select an appropriate movie clip, Timeline, or screen as the parent for the RPC component. The component's storage will become available for garbage collection when the parent goes away.
- In ActionScript, you can assign null to this property at any time.

RPC.send

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("send", myListenerObject);
```

Description

Event; broadcasts during the processing of a `trigger()` operation, 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 using the `params` property.

RPC.status

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("status", myListenerObject);
```

Description

Event; broadcasts 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 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 (a) the `trigger()` function is called, and (b) multipleSimultaneousAllowed is false, and (c) a call is already in progress. After this event occurs, the attempted call is considered complete, and there will be no "result" or "send" event.
InvalidParams	no data	This event is emitted if the `trigger()` function found that the "params" property did not contain valid data. If the "suppressInvalidCalls" property is true, then the attempted call is considered complete, and there will be no "result" or "send" event.

RPC.suppressInvalidCalls

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.suppressInvalidCalls;
```

Description

Property; indicates whether to suppress a call if parameters are invalid. If true, then the `trigger()` function will not perform a call if the bound parameters fail the validation. A "status" event will be emitted, with the code `InvalidParams`. If false, then the call will take place, using the invalid data as required.

RPC.trigger()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.trigger();
```

Description

Method; initiates an Remote Procedure Call. Each RPC component defines exactly what this involves. If the operation is successful, the results of the operation will 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 execute 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 via the connection method indicated (for example, HTTP).

Screen class (Flash Professional only)

Inheritance UIObject > UIComponent > View > Loader > Screen

ActionScript Class Name mx.screens.Screen

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 "Working with Screens (Flash Professional Only)" in Using Flash Help.

The Screen class has two primary subclasses: Slide and Form.

The Slide class provides the runtime behavior for slide presentations. The Slide class contains built-in navigation and sequencing capabilities, as well as the ability to easily attach transitions between slide using Behaviors. Slide objects maintain a notion of "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 479.

The Form class provides the runtime environment for form applications. Forms have the ability to overlay and be containers for, or be contained by, other components. Unlike slides, forms don't provide any sequencing or navigation capabilities. For more information on using the Form class, see "Form class (Flash Professional only)" on page 277.

The Screen class provides functionality common to both slides and forms.

Screens know how to manage their children Every screen comes with a built-in property that is a collection of screens as children. This collection is determined by the layout of the screen hierarchy in the Screens Pane. Screens can have any number of children (including zero), which themselves can have children.

Screens can hide/show their children Because screens are, 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 For example, you can trigger a sound to play, or start playing some video, when a particular screen becomes visible.

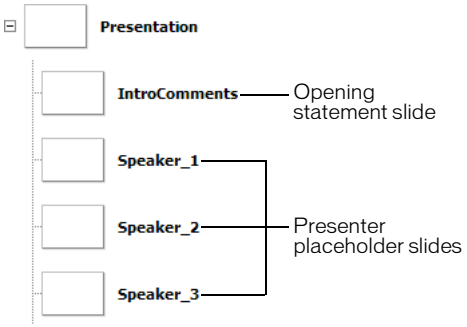
Loading external content into screens (Flash Professional only)

The Screen class extends the Loader class (see "Loader component" on page 314), which provides the ability to easily manage and load external SWFs (and JPEGs). The Loader class contains a property called `contentPath` which specifies the URL of an external SWF or JPEG, or the linkage identifier of a movie clip in the Library.

Using this feature, you can load external screen tree (or any external SWF) as a child of any screen node. This provides a useful way of modularizing your screens-based movies and dividing them into multiple, separate SWFs.

For example, suppose you had a slide presentation in which three people were each contributing a single section. You could ask each presenter to each create a separate slide presentation (SWF). You would then create a "master slide presentation" that contained three placeholder slides, one for each slide presentation being created by the presenters. For each placeholder slide, you could point its `contentPath` property to each of the SWFs.

For example, the “master” slide presentation could be arranged as shown in the following illustration:



“Master” SWF slide presentation structure

Suppose that presenters have provided you with three SWFs, speaker_1.swf, speaker_2.swf, and speaker_3.swf. You could easily assemble the overall presentation by pointing `contentPath` property for each placeholder slide, either using ActionScript, or by setting the Property inspector’s `contentPath` property for each slide.

```
Speaker_1.contentPath = speaker_1.swf;
Speaker_2.contentPath = speaker_2.swf;
Speaker_3.contentPath = speaker_3.swf;
```

You can also set the `contentPath` property for each slide using the Property inspector. Note that, by default, when you set a slide’s `contentPath` in the Property inspector (or with code, as shown above), the specified SWF will load as soon as the “master presentation” SWF has loaded. To reduce initial load time, consider setting the contentPath property within an on(reveal) handler attached to each slide.

```
// Attached to Speaker_1 slide
on(reveal) {
  this.contentPath="speaker_1.swf";
}
```

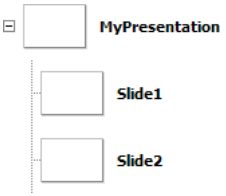
Alternatively, you could set to `false` the slide’s `autoLoad` property (inherited from the Loader class), and then call the `load()` method on the slide (also inherited from the Loader class) when the slide has been revealed.

```
// Attached to Speaker_1 slide
on(reveal) {
  this.load();
}
```

Referencing loaded screens with ActionScript

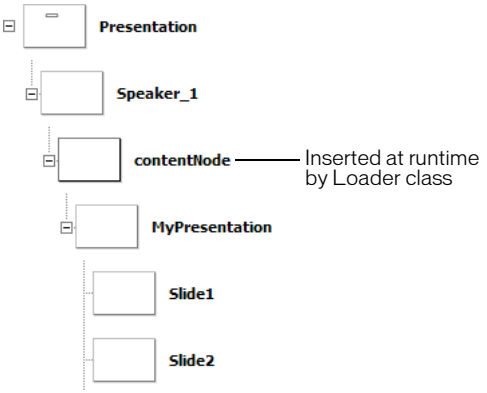
The Loader class creates an internal movie clip named `contentNode` into which it loads the SWF or JPEG specified by the `contentPath` property. This movie clip, in effect, adds an extra screen node between the “placeholder” slide (that you created in the “master” presentation above) and the first slide in the loaded slide presentation.

For example, suppose that the SWF 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 is loaded into the placeholder slide, the overall slide presentation would now 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, referring to the illustration above, the slide named MyPresentation (and its subslides) is part of the contiguous slide tree rooted at the Presentation slide, and is not treated as a separate subtree.

Method summary for the Screen class

Method	Description
`Screen.getChildScreen()`	Returns the child screen of this screen at a particular index.

Inherits all methods from UIObject, UIComponent, View, and Loader component.

Property summary for the Screen class

Property	Description
`Screen.currentFocusedScreen`	Returns the screen that contains the global current focus.
`Screen.indexInParent`	Returns the screen's index (zero-based) in its parent screen's list of child screens.
`Screen.numChildScreens`	Returns the number of child screens contained by the screen.
`Screen.parentIsScreen`	Returns a Boolean (`true` or `false`) value that indicates whether the screen's parent object is itself a screen.
`Screen.rootScreen`	Returns the root screen of the (sub)-tree that contains the screen.

Inherits all properties from UIObject, UIComponent, View, and Loader component.

Event summary for 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.

Inherits all events from UIObject, UIComponent, View, and Loader component.

Screen.allTransitionsInDone

Availability

Flash Player 6 version 79.

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 *`myScreen`*.

Example

In the following example, a button (`nextSlide_btn`) that's contained by the slide named `mySlide` is made visible once all the "in" transitions applied to `mySlide` have completed.

```
// Attached to mySlide:
on(allTransitionsInDone) {
  this.nextSlide_btn._visible = true;
}
```

See also

`Screen.allTransitionsOutDone`

Screen.allTransitionsOutDone

Availability

Flash Player 6 version 79.

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 *`myScreen`*.

See also

`Screen.currentFocusedScreen`

Screen.currentFocusedScreen

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
mx.screens.Screen.currentFocusedScreen
```

Description

Static property (read-only); returns a reference to the "leafmost" Screen object that contains the global current focus. The focus may be on the screen itself, or on a movie clip, text object, or component inside that screen. Defaults to `null` if there is no current focus.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

myScreen`.getChildScreen(`*index*`)`

Parameters

childIndex A number that indicates the index (zero-based) of the child screen to return.

Returns

A Screen object.

Description

Method; returns the child Screen object of *myScreen* whose index is *childIndex*.

Example

The following example displays in the Output panel the names of all the child screens belonging to the root screen named `Presentation`.

```
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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myScreen.indexInParent
```

Description

Property (read-only); contains the index (zero-based) of *myScreen* in its parent's list of subscreens.

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 version 79.

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 was 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 a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
on(mouseDown) {
  // 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 a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

Screen.mouseMove

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
on(mouseDown) {
  // 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 only sent 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 a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

Note: Use of this event may impact system performance and should be used judiciously.

Screen.mouseOut

Availability

Flash Player 6 version 79.

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 a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

Note: Use of this event may impact system performance and should be used judiciously.

Screen.mouseOver

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
on(mouseDown) {
  // 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 to inside its bounding box.

When the event is triggered, it automatically passes an event object (*eventObj*) to the handler. Each event object has a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

Note: Use of this event may impact system performance and should be used judiciously.

Screen.mouseUp

Availability

Flash Player 6 version 79.

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 a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

Screen.mouseUpSomewhere

Availability

Flash Player 6 version 79.

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 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 a set of properties that contain information about the event. You can use these properties to write code that handles the event. For more information about event objects, see "Event Objects" on page 562.

Screen.numChildScreens

Availability

Flash Player 6 version 79.

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 belonging 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 version 79.

Edition

Flash MX Professional 2004.

Usage

myScreen.parentIsScreen

Description

Property (read-only): returns a Boolean (`true` or `false`) value indicating whether the specified screen's parent object is also a screen (`true`), or not (`false`). If false, then *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 so, it's assumed that `myScreen` is the root, or master, slide in the presentation and therefore has no sibling slides. Otherwise, if `myScreen.parentIsScreen` is `true`, then the number of `myScreen`'s sibling slides is displayed in the Output panel.

```
if (myScreen.parentIsScreen) {
  trace("I have "+myScreen._parent.numChildScreens+" sibling screens");
} else {
  trace("I am the root screen and have no siblings");
}
```

Screen.rootScreen

Availability

Flash Player 6 version 79.

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

```
var myRoot:mx.screens.Screen = myScreen.rootScreen;
```

ScrollPane component

The Scroll Pane component displays movie clips, JPEG files, and SWF files in a scrollable area. You can enable scroll bars to display images in a limited area. You can display content that is loaded from a local location, or from over the internet. You can set the content for the scroll pane both while authoring and at runtime using ActionScript.

Once the scroll pane has focus, if the content of the scroll pane has valid tab stops, those markers will 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	Content moves up one vertical line scroll.
End	Content moves to the bottom of the scroll pane.
Left	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	Content moves left one horizontal line scroll
Up	Content moves down one vertical line scroll.

For more information about controlling focus, see "Creating custom focus navigation" on page 24 or "FocusManager class" on page 270.

A live preview of each ScrollPane instance reflects changes made to parameters in the Property inspector or Component Inspector panel while 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

The following are authoring parameters that you can set for each ScrollPane component instance in the Property inspector or in the Component Inspector panel:

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 pressed. The default value is 5.

hPageScrollSize indicates the number of units a horizontal scroll bar moves each time the track is pressed. 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 allows a user to scroll the content within the scroll pane (true) or not (false). The default value is false.

vLineScrollSize indicates the number of units a vertical scroll bar moves each time an arrow button is pressed. The default value is 5.

vPageScrollSize indicates the number of units a vertical scroll bar moves each time the track is pressed. 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 ScrollPane components using its properties, methods, and events. For more information, see ScrollPane class.

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, do the following:

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`, that has already been placed on Stage. 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 both 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. See ScrollPane class. If the ScrollPane is too small, the content may not display correctly.

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.

When the scroll pane is resized, the buttons remain the same size and the scroll track and thumb expand or contract, and their hit areas are resized.

Using styles with the ScrollPane component

The ScrollPane doesn't support styles, but the scroll bars that it uses do.

Using skins with the ScrollPane component

The ScrollPane component doesn't have any skins of its own, but the scroll bars that it uses do have skins.

ScrollPane class

Inheritance UIObject > UIComponent > View > ScrollView > ScrollPane

ActionScript Class Name mx.containers.ScrollPane

The properties of the ScrollPane class allow you to set the content, monitor the loading progress, and adjust the scroll amount at runtime.

Setting a property of the ScrollPane class with ActionScript overrides the parameter of the same name set in the Property inspector or Component Inspector panel.

You can set up a scroll pane to allow users to drag the content within the pane by setting 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 only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.containers.ScrollPane.version);
```

Note: The following code returns undefined: `trace(myScrollPaneInstance.version);`.

Method summary for the ScrollPane class

Method	Description
`ScrollPane.getBytesLoaded()`	Returns the number of bytes of content loaded.
`ScrollPane.getBytesTotal()`	Returns the total number of content bytes to be loaded.
`ScrollPane.refreshPane()`	Reloads the contents of the scroll pane.

Inherits all methods from UIObject and UIComponent.

Property summary for 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 an arrow button is pressed.
`ScrollPane.hPageScrollSize`	The amount of content to scroll horizontally when the track is pressed.
`ScrollPane.hPosition`	The horizontal pixel position of the scroll pane.
`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 there is scrolling when a user presses and drags within the ScrollPane (`true`) or not (`false`). The default value is `false`.
`ScrollPane.vLineScrollSize`	The amount of content to scroll vertically when an arrow button is pressed.
`ScrollPane.vPageScrollSize`	The amount of content to scroll vertically when the track is pressed.
`ScrollPane.vPosition`	The vertical pixel position of the scroll pane.
`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"`.

Inherits all properties from UIObject and UIComponent.

Event summary for the ScrollPane class

Method	Description
`ScrollPane.complete`	Broadcast when the scroll pane content is loaded.
`ScrollPane.progress`	Broadcast while the scroll bar content is loading.
`ScrollPane.scroll`	Broadcast when the scroll bar is pressed.

Inherits all events from UIObject and UIComponent.

ScrollPane.complete

Availability

Flash Player 6 version 79.

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 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 ScrollPane component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 version 79.

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 that loads the content.

If you load content using a relative URL, the loaded content must be relative to the location of the SWF 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 with 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 version 79.

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 an instance of the ScrollPane class called `scrollPane`. It then defines a listener object called `loadListener` with a `progress` event handler that calls the `getBytesLoaded()` method 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 just commences
  {
    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 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

scrollPaneInstance.hLineScrollSize

Description

Property; the number of pixels to move the content when the left or right arrow in the horizontal scroll bar is pressed. The default value is 5.

Example

This example increases the horizontal scroll unit to 10:

```
scrollPane.hLineScrollSize = 10;
```

ScrollPane.hPageScrollSize

Availability

Flash Player 6 version 79.

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 pressed. 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 version 79.

Edition

Flash MX 2004.

Usage

scrollPaneInstance.hPosition

Description

Property; the pixel position of the horizontal scroll bar. The 0 position is to the left of the bar.

Example

This example sets the scroll bar to 20:

```
scrollPane.hPosition = 20;
```

ScrollPane.hScrollPolicy

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

scrollPaneInstance.hScrollPolicy

Description

Property; determines whether the horizontal scroll bar is always present (`"on"`), 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 version 79.

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. This event is triggered when the content starts loading by setting the value of `contentPath` property.

The first usage example uses an `on()` handler and must be attached directly to a ScrollPane 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

The following code creates a ScrollPane instance called `scrollPane` and then creates a listener object with an event handler for the `progress` event that sends a message to the Output panel about what number of bytes of the 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 version 79.

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 contents. You could use this method if, for example, you've loaded a form into a ScrollPane and an input property (for example, in a text field) has been changed using ActionScript. 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 version 79.

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 is a `type` property defined for the `scroll` event, the value is `"scroll"`. There is also a `direction` property with the possible values `"vertical"` and `"horizontal"`.

Description

Event; broadcast to all registered listeners when a user presses the scroll bar buttons, thumb, or track. Unlike other events, the `scroll` event is broadcast when a user presses on the scroll bar and continues broadcasting until the scroll bar is released.

The first usage example uses an `on()` handler and must be attached directly to a ScrollPane 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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, as in the following:

```
spInstance.contentPath = "mouse3.jpg";
form = new Object();
form.scroll = function(eventObj){
  trace("ScrollPane scrolled");
}
spInstance.addEventListener("scroll", form);
```

See also

`UIEventDispatcher.addEventListener()`

ScrollPane.scrollDrag

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`scrollPaneInstance``.scrollDrag`

Description

Property; a Boolean value that indicates whether there is scrolling when a user presses and drags within the ScrollPane (`true`) or not (`false`). The default value is `false`.

Example

This example enables mouse scrolling within the scroll pane:

```
scrollPane.scrollDrag = true;
```

ScrollPane.vLineScrollSize

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`scrollPaneInstance``.vLineScrollSize`

Description

Property; the number of pixels to move the display area when the up or down arrow button in a vertical scroll bar is pressed. The default value is 5.

Example

This code increases the amount that the display area moves when the vertical scroll bar arrow buttons are pressed to 10:

```
scrollPane.vLineScrollSize = 10;
```

ScrollPane.vPageScrollSize

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.vPageScrollSize
```

Description

Property; the number of pixels to move the display area when the track in a vertical scroll bar is pressed. The default value is 20.

Example

This code increases the amount that the display area moves when the vertical scroll bar arrow buttons are pressed to 30:

```
scrollPane.vPageScrollSize = 30;
```

ScrollPane.vPosition

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.vPosition
```

Description

Property; the pixel position of the vertical scroll bar. The default value is 0.

ScrollPane.vScrollPolicy

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

```
scrollPaneInstance.vScrollPolicy
```

Description

Property; determines whether the vertical scroll bar is always present (`"on"`), 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";
```

Slide class (Flash Professional only)

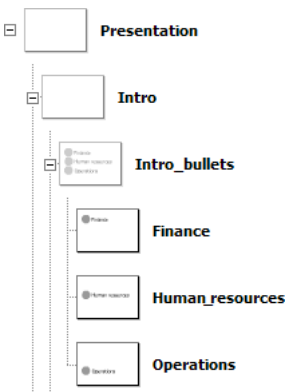
Inheritance UIObject > UIComponent > View > Loader > Screen > Slide

ActionScript Class Name mx.screens.Slide

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 "Working with Screens (Flash Professional Only)" in Using Flash Help.

The Slide class extends the Screen class (see "Screen class (Flash Professional only)" on page 452), 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 a notion of "state", so the user can advance to the next or previous slide in a presentation and when the next slide in a presentation is shown, the previous slide is hidden.

Note that you can only navigate to (or "stop on") slides that don't contain any child slides, or "leaf" slides. For example, the following illustration 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 (see "Loader class" on page 316), which lets you easily load external SWFs (or JPEGs) into a given slide. This provides a way to modularize your slide presentations, and reduce initial download time. For more information, see "Loading external content into screens (Flash Professional only)" on page 452.

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 the 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 one of the built-in behaviors for controlling slide presentations that are available in the Behaviors panel (Window > Development Panels > Behaviors). For more information on using behaviors with slides, see "Adding controls to screens using behaviors (Flash Professional only)" in Using Flash Help.

Slide parameters

The following are authoring parameters that you can set for each slide in the Property inspector or in the Component Inspector panel:

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 for a SWF or JPG file to load into the slide. By default, loaded content clips to fit the slide.

overlayChildren specifies whether the slide's child slides remain visible when you navigate from one child slide to the next (true), or not (false).

playHidden specifies whether the slide continues to play when hidden (true) or not (false).

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. Note that the Behavior panel (Window > Development Panels > Behaviors) contains several behaviors for creating slide navigation. In this example, you write your own ActionScript to create Next and Previous buttons for a slide presentations.

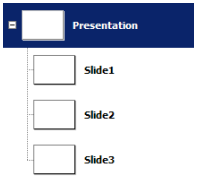
To create a slide presentation with navigation:

1 In Flash, select File > New.

2 Click the General tab and select Flash Slide Presentation under Type.

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 (Window > Development Panels > Components).

7 Drag a Button component from the Components panel to the bottom of the Stage.

8 In the Property inspector (Window > Properties) type "Next Slide" for the Button component's Label property.

9 Open the Actions panel, if it's not already open, by selecting Window > Development Panels > Actions.

10 Type the following code in the Actions panel:

```
on(click) {
  _parent.currentSlide.gotoNextSlide();
}
```

11 Test the SWF (Control > Test Movie) and click the Next Slide button to advance to the next slide.

Method summary for the Slide class

Property	Description
`Slide.getChildSlide()`	Returns the child slide of this slide at a given index.
`Slide.gotoFirstSlide()`	Navigates to the first leaf node in the slide's hierarchy of subslides.
`Slide.gotoLastSlide()`	Navigates to the last leaf node in the slide's hierarchy of subslides.
`Slide.gotoNextSlide()`	Navigates to the next slide.
`Slide.gotoPreviousSlide()`	Navigates to the next slide.
`Slide.gotoSlide()`	Navigates to an arbitrary slide.

Inherits all methods from UIObject, UIComponent, View, Loader component, and Screen class (Flash Professional only).

Property summary for the Slide class

Property	Description
`Slide.autoKeyNav`	Determines whether or not the slide uses default keyboard handling to navigate to the next/previous slide.
`Slide.currentSlide`	Returns the immediate child of the slide that contains the currently active slide.
`Slide.currentSlide`	Returns the currently active slide.
`Slide.currentFocusedSlide`	Returns the "leafmost" slide that contains the global current focus.
`Slide.defaultKeydownHandler`	Callback handler that overrides the default keypress slide navigation (left and right arrow).
`Slide.firstSlide`	Returns the slide's first child slide that has no children.
`Slide.getChildSlide()`	Returns the child slide at a specified index.
`Slide.indexInParentSlide`	Returns the slide's index (zero-based) in its parent's list of subslides.
`Slide.lastSlide`	Returns the slide's last child slide that has no children.
`Slide.nextSlide`	Returns the next leaf node slide.
`Slide.numChildSlides`	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`	Returns a Boolean value indicating whether the parent object of the slide is also a slide (`true`) or not (`false`).
`Slide.playHidden`	Determines whether or not the slide continues to play when hidden.
`Slide.previousSlide`	Returns the previous leaf node slide.
`Slide.revealChild`	Returns the root of the slide tree that contains the slide.

Inherits all properties from UIObject, UIComponent, View, Loader component, and Screen class (Flash Professional only).

Event summary for the Slide class

Event	Description
`Slide.hideChild`	Broadcast when all children of a slide changes from visible to invisible.
`Slide.revealChild`	Broadcast when all children of a slide changes from invisible to visible.

Inherits all events from UIObject, UIComponent, View, Loader component, and Screen class (Flash Professional only).

Slide.autoKeyNav

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

mySlide.autoKeyNav

Description

Property; determines whether or not the slide uses default keyboard handling to navigate to the next/previous slide when *mySlide* has focus. This property accepts one of the following string values: "true", "false", or "inherit". You can also override this default keyboard handling behavior using the Slide.defaultKeydownHandler property.

You can also set this property using the Property inspector.

When set to "true", pressing the right arrow (Key.RIGHT) or the Spacebar (Key.SPACE) when *mySlide* has focus advances to the next slide; pressing the left arrow (Key.Left) moves to the previous slide.

When set to "false", no default keyboard handling takes place when *mySlide* has focus.

When set to "inherit", *mySlide* checks the autoKeyNav property of its parent slide. If the parent of *mySlide* is also set to "inherit", then *mySlide*'s parent's parent is examined, and so on, until a parent slide is found whose autoKeyNav property is set to "true" or "false".

If *mySlide* has no parent slide (that is, if (mySlide.parentIsSlide == false) is true) then 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.currentSlide

Availability

Flash Player 6 version 79.

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 pressed.

```
// Attached to button instance contained by Presentation slide:
on(press) {
  _parent.currentSlide.gotoNextSlide();
}
```

See also

`Slide.gotoNextSlide()`

Slide.currentChildSlide

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
mx.screens.Slide.currentFocusedSlide
```

Description

Property (read-only); returns “leaf-most” slide 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; returns `null` if there is no current focus.

Example

```
var focusedSlide = mx.screens.Slide.currentFocusedSlide;
```

Slide.defaultKeydownHandler

Availability

Flash Player 6 version 79.

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 the value returned by Key.getCode().
- `shiftKey` A Boolean (true or false) value indicating if the Shift key is currently being pressed (true) or not (false).
- `ctrlKey` A Boolean (true or false) value indicating if the Control key is currently being pressed (true) or not (false).

Returns

Nothing.

Description

Callback handler; lets you override the default key board 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`.

Automatic keyboard handling is enabled when the current slide's `Slide.autoKeyNav` property is set to "true", or if it set to "inherit" and the most immediate ancestor of the current slide that is not "inherit" is either the root slide of the presentation, or whose `autoKeyNav` value is set to "true".

If automatic keyboard handling is enabled for the current slide,

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/down arrow for navigation instead of left/right arrow.

```
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 version 79.

Edition

Flash MX Professional 2004.

Usage

mySlide.firstSlide

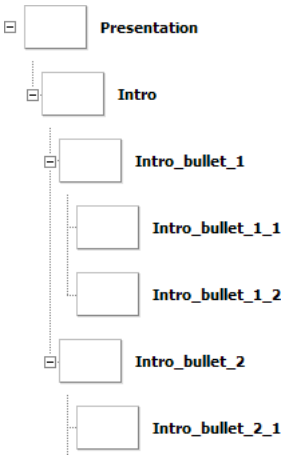
Description

Property (read-only); returns the first child slide of *mySlide* that has no child slides.

Example

For example, in the hierarchy of slides shown below, the following statements are all true:

```
Presentation.Intro.firstSlide == Intro_bullet_1_1;
Presentation.Intro_bullet_1.firstSlide == Intro_bullet_1-1;
```



Slide.getChildSlide()

Availability

Flash Player 6 version 79.

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*. This method is useful, for example, to iterate over a set of child slides whose indices are known, as the following example shows.

Example

This example displays in the Output panel 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.gotoSlide()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoSlide(`*`newSlide`*`)`

Parameters

`newSlide` The slide to navigate to.

Returns

A Boolean value (`true` or `false`) 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`, then the following `gotoSlide()` method 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`, then 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 `gotoSlide()` method to send the presentation to the next slide in the presentation.

```
on(click) {
  _parent.gotoSlide(_parent.currentSlide.nextSlide);
}
```

Note that 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.gotoFirstSlide()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoFirstSlide()`

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`

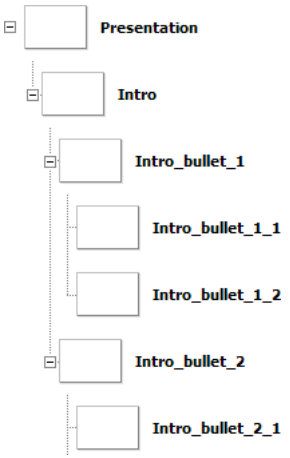
Example

In the slide hierarchy illustrated below, the following method calls would all navigate to the slide named `Intro_bullet_1_1`.

```
Presentation.gotoFirstSlide();
Presenation.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 version 79.

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoLastSlide()`

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.

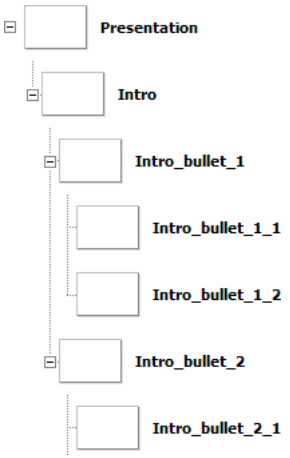
Example

In the slide hierarchy illustrated below, the following method calls would navigate to the slide named `Intro_bullet_1_2`.

```
Presenation.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 version 79.

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoNextSlide()`

Returns

A Boolean (`true` or `false`) value, or `null`; returns `true` if the method successfully navigated to the next slide; returns `false` if the presentation is already at the last slide when the method is invoked (that is, if `currentSlide.nextSlide` is `null`); 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.

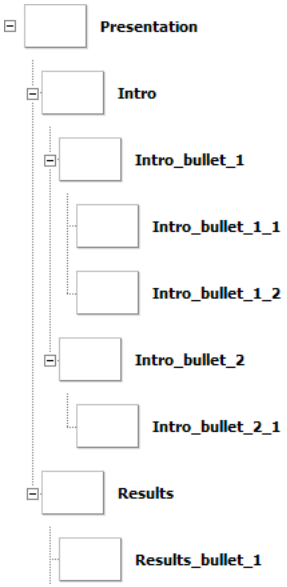
Typically, `gotoNextSlide()` is called on the leaf node that represents the current slide. If called on a non-leaf node, `someNode`, then `someNode.gotoNextSlide()` advances to the first leaf node in the next slide or "section". For more information, see the example below.

This method has no effect when invoked on a slide that does not contain the current slide (see example below).

Also, this method 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, invoking `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()` will have no effect, since `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 version 79.

Edition

Flash MX Professional 2004.

Usage

`mySlide``.gotoPreviousSlide()`

Returns

A Boolean (`true` or `false`) value, or `null`; returns `true` if the method successfully navigated to the previous slide; returns `false` if the presentation is at the first slide when the method is invoked (that is, if `currentSlide.nextSlide` is `null`); 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, 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.

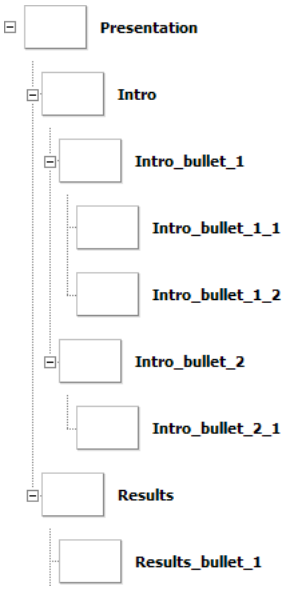
Typically, `gotoPreviousSlide()` is called on the leaf node that represents the current slide. If called on a non-leaf node, `someNode`, then `someNode.gotoPreviousSlide()` advances to the first leaf node in the next slide or "section". For more information, see the example below.

This method has no effect when invoked on a slide that does not contain the current slide (see example below).

Also note that this method 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 naviagation.

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, invoking `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()` will 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.hideChild

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
on(hideChild) {
  // your code here
}
```

Description

Event; broadcasted each time a child of a slide object changes visible to non-visible. This event is only broadcasted by slide objects, not Form objects. The main use of the `hideChild` event is to apply "out" transitions to all the children 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 belonging to the root slide, as it appears.

```
on(revealChild) {
  var child = eventObj.target._name;
  trace(child + " has just appeared");
}
```

See also

`Slide.revealChild`

Slide.indexInParentSlide

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

`mySlide``.indexInParent`

Description

Property (read-only); returns the index (zero-based) of *`mySlide`* in its parent's list of child slides.

Example

The following code uses the `indexInParent` 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 one (`currentSlide.indexInParent+1`) to display more meaningful values.

See also

`Slide.numChildSlides`, `Slide.revealChild`

Slide.lastSlide

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

`mySlide``.lastSlide`

Description

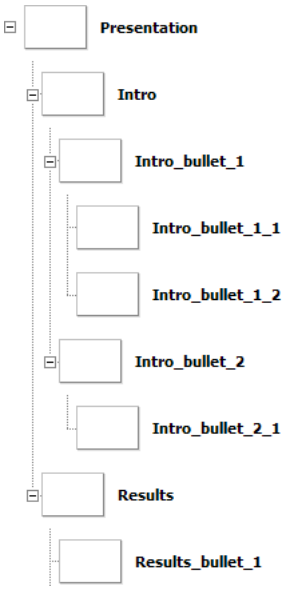
Property (read-only); returns the last child slide of *`mySlide`* that has no child slides.

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 version 79.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

mySlide.numChildSlides

Description

Property (read-only); returns the number of child slides that *mySlide* contains. Note that 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.numChildSlide and the Slide.getChildSlide() method to iterate over all the child slides of the root Presentation slide and 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 version 79.

Edition

Flash MX Professional 2004.

Usage

mySlide.overlayChildren

Description

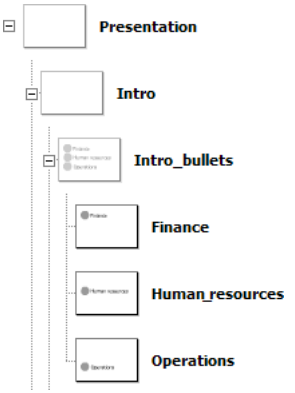
Property; determines whether or not child slides of *mySlide* remain visible when navigating from one child slide to the next. When set to `true`, the previous slide remains visible when control passes to its next sibling slide; when set to `false`, the previous slide is invisible when control passes 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

For 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 will remain on the Stage as the other bullets points appear.



Slide.parentIsSlide

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

mySlide`.parentIsSlide`

Description

Property (read-only); a Boolean (`true` or `false`) value indicating whether the parent object of *mySlide* is also a Slide. If the parent object of *mySlide* is a Slide, or a subclass of Slide, then this property will return `true`; otherwise, it returns false.

If *mySlide* is the root slide in a presentation then this property will return `false` since the Presentation slide's parent is the main Timeline (_level0), not a slide. This property will also return `false` if a form is the parent of *mySlide*.

Example

The following code determines if the parent object of the slide `mySlide` is itself a slide. If so, it's assumed that `mySlide` is the root, or master, slide in the presentation and therefore has no sibling slides. Otherwise, if `mySlide.parentIsSlide` is `true`, and the number of `mySlide`'s sibling slides is displayed in the Output panel.

```
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 version 79.

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`* will continue to play when hidden. When set to `false`, *`mySlide`* is stopped upon being hidden; upon being revealed play restarts at Frame 1 of *`mySlide`*.

You can also set this property in the Property inspector of the Flash authoring environment.

Slide.previousSlide

Availability

Flash Player 6 version 79.

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`—then 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
on(revealChild) {
  // your code here
}
```

Description

Event; broadcasted each time a child slide of a slide object changes non-visible 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 version 79.

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, please refer to the W3C web site.

Method summary for the StyleManager class

Method	Description
StyleManager.registerColorName()	Registers a new color name with the StyleManager.
StyleManager.registerColorStyle()	Registers a new color style with the StyleManager.
StyleManager.registerInheritingSyle()	Registers a new inheriting style with the StyleManager.

StyleManager.registerColorName()

Availability

Flash Player 6 version 79.

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 StyleManager.

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 version 79.

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 StyleManager.

Example

The following example registers "highlightColor" as a color style:

```
StyleManager.registerColorStyle("highlightColor");
```

StyleManager.registerInheritingSyle()

Availability

Flash Player 6 version 79.

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 styles properties to inheriting.

Example

The following example registers `newProp1` as an inheriting style:

```
StyleManager.registerInheritingStyle("newProp1");
```

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.

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	Moves 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 24 or "FocusManager class" on page 270.

A live preview of each TextArea instance reflects changes made to parameters in the Property inspector or Component Inspector panel while 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 into 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" on page 516. For example, you could use a TextArea component as a comment field in a form. You could set up a listener that checks if 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 component parameters

The following are authoring parameters that you can set for each TextArea component instance in the Property inspector or in the Component Inspector panel:

text indicates the contents of the TextArea. You cannot enter carriage returns in the Property inspector or Component Inspector panel. The default value is "" (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 TextArea components using its properties, methods, and events. For more information, see TextArea class.

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, do the following:

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 in 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 both 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 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 non-color style properties. For more information, see “Using styles to customize component color and text” on page 27.

A TextArea component supports the following styles:

Style	Description
`color`	The default color for text.
`embedFonts`	The fonts to embed in the document.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.
`fontStyle`	The font style, either "normal",or "italic".
`fontWeight`	The font weight, either "normal" or "bold".

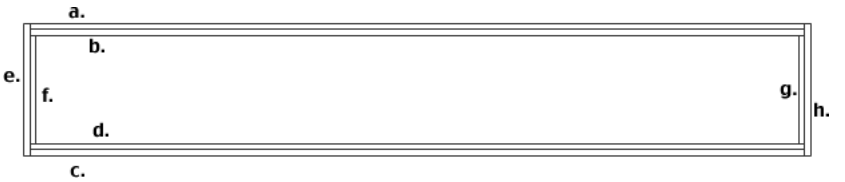
Style	Description
`textAlign`	The text alignment: either "left", "right", or "center".
`textDecoration`	The text decoration, either "none" or "underline".

Using skins with the TextArea component

The TextArea component uses the RectBorder class to draw its border. You can use the `setStyle()` method (see `UIObject.setStyle()`) to change the following RectBorder style properties:

RectBorder styles	Letter
`borderColor`	a
`highlightColor`	b
`borderColor`	c
`shadowColor`	d
`borderCapColor`	e
`shadowCapColor`	f
`shadowCapColor`	g
`borderCapColor`	h

The style properties set the following positions on the border:



TextArea class

Inheritance UIObject > UIComponent > View > ScrollView > TextArea

ActionScript Class Name mx.controls.TextArea

The properties of the TextArea class allow you to 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 panel.

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 only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.TextArea.version);
```

Note: The following code returns undefined: `trace(myTextAreaInstance.version);`.

Property summary for 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 within the scroll pane.
`TextArea.hScrollPolicy`	Indicates whether the horizontal scroll bar is always on (`"on"`), never on (`"off"`), or turns on when needed (`"auto"`).S
`TextArea.html`	A flag that indicates whether the text field can be formatted with HTML.
`TextArea.length`	The number of characters in the text field. This property is read-only.
`TextArea.maxChars`	The maximum number of characters that the text field can contain.
`TextArea.maxHPosition`	The maximum value of `TextArea.hPosition`.
`TextArea.maxVPosition`	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 into the text field.
`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"`), never on (`"off"`), or turns on when needed (`"auto"`).S
`TextArea.wordWrap`	A Boolean value indicating whether the text wraps (`true`) or not (`false`).

Event summary for the TextArea class

Event	Description
`TextArea.change`	Notifies listeners that text has changed.

TextArea.change

Availability

Flash Player 6 version 79.

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 prevent certain characters from being added to the component's text field; instead, use `TextArea.restrict`.

The first usage example uses an `on()` handler and must be attached directly to a TextArea 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example traces the total of number of times the text field has changed:

```
myTextArea.changeHandler = function(obj) {
  this.changeCount++;
  trace(obj.target);
  trace("text has changed " + this.changeCount + " times now! it now contains
  " +
this.text);
}
```

See also

`UIEventDispatcher.addEventListener()`

TextArea.editable

Availability

Flash Player 6 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

`textAreaInstance``.hPosition`

Description

Property; defines the horizontal position of the text within the field. The default value is 0.

Example

The following code displays the left-most characters in the field:

```
myTextArea.hPosition = 0;
```

TextArea.hScrollPolicy

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

textAreaInstance.hScrollPolicy

Description

Property; determines whether the horizontal scroll bar is always present ("on"), 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 version 79.

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 version 79.

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 version 79.

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 `maxChars` property only indicates 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 version 79.

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 scrolls the text to the far right:

```
myTextArea.hPosition = myTextArea.maxHPosition;
```

See also

TextArea.vPosition

TextArea.maxVPosition

Availability

Flash Player 6 version 79.

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 scrolls the text to the bottom of the component:

```
myTextArea.vPosition = myTextArea.maxVPosition;
```

See also

TextArea.hPosition

TextArea.password

Availability

Flash Player 6 version 79.

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 the value of `password` is `true`, the text field is a password text field and hides the input characters. If `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 version 79.

Edition

Flash MX 2004.

Usage

textAreaInstance.restrict

Description

Property; indicates the set of characters that a user may enter into the text field. The default value is undefined. If the value of the restrict property is null, a user can enter any character. If the value of the `restrict` property is an empty string, no characters may be entered. If the value of the `restrict` property is a string of characters, you can enter only characters in the string into the text field; the string is scanned from left to right. A range may be specified using the dash (-).

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.

If the string begins with "^", all characters are initially accepted and succeeding characters in the string are excluded from the set of accepted characters. If the string does not begin with "^", no characters are initially accepted and succeeding characters in the string are included in the set of accepted characters.

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.text

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

textAreaInstance.text

Description

Property; the text contents of a TextArea component. The default value is "" (empty string).

Example

The following code places a string in the myTextArea instance 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 version 79.

Edition

Flash MX 2004.

Usage

textAreaInstance.vPosition

Description

Property; defines the vertical position of text in a text field. The scroll 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 makes the topmost characters in a field display:

```
myTextArea.vPosition = 0;
```

TextArea.vScrollPolicy

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

textAreaInstance.vScrollPolicy

Description

Property; determines whether the vertical scroll bar is always present ("on"), 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 version 79.

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 is a single-line component that wraps the native ActionScript TextField object. You can use styles to customize the TextInput component; when an instance is disabled its contents display 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	Moves character 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 24 or "FocusManager class" on page 270.

A live preview of each TextInput instance reflects changes made to parameters in the Property inspector or Component Inspector panel while authoring. Text is not selectable in the live preview and you cannot enter text into 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" on page 504. For example, you could use a TextInput component as a password field in a form. You could set up a listener that checks if 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 component parameters

The following are authoring parameters that you can set for each TextInput component instance in the Property inspector or in the Component Inspector panel:

text specified the contents of the TextInput. You cannot enter carriage returns in the Property inspector or Component Inspector panel. The default value is "" (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 TextInput components using its properties, methods, and events. For more information, see TextInput class.

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 have been entered.

To create an application with the TextInput component, do the following:

1 Drag a TextInput component from the Components panel to the Stage.

2 In the Property inspector, enter the instance name **passwordField**.

3 In the Property inspector, do the following:

- Leave the text parameter blank.
- Set the editable parameter to true.
- Set the password parameter to true.

4 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.

5 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 both 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 on the instance.

A TextInput component supports the following styles:

Style	Description
`color`	The default color for text.
`embedFonts`	The fonts to embed in the document.
`fontFamily`	The font name for text.
`fontSize`	The point size for the font.

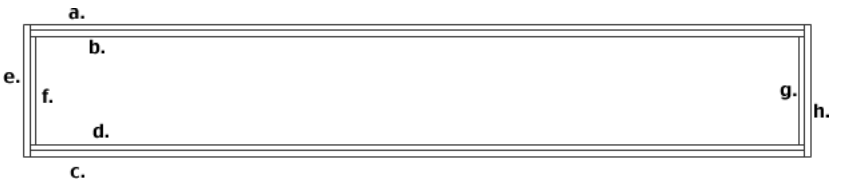
Style	Description
fontStyle	The font style, either "normal",or "italic".
fontWeight	The font weight, either "normal" or "bold".
textAlign	The text alignment: either "left", "right", or "center".
textDecoration	The text decoration, either "none" or "underline".

Using skins with the TextInput component

The TextArea component uses the RectBorder class to draw its border. You can use the setStyle() method (see UIObject.setStyle()) to change the following RectBorder style properties:

RectBorder styles	Letter
borderColor	a
highlightColor	b
borderColor	c
shadowColor	d
borderCapColor	e
shadowCapColor	f
shadowCapColor	g
borderCapColor	h

The style properties set the following positions on the border:



TextInput class

Inheritance UIObject > UIComponent > TextInput

ActionScript Class Name mx.controls.TextInput

The properties of the TextInput class allow you to 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 panel.

The TextInput component uses the FocusManager 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 270.

The TextInput component supports CSS styles and any additional HTML styles supported by Flash Player. For information about CSS support, see the W3C specification.

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 only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.controls.TextInput.version);
```

Note: The following code returns undefined: `trace(myTextInputInstance.version);`.

Method summary for the TextInput class

Inherits all methods from UIObject and UIComponent.

Property summary for 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 field.
`TextInput.length`	The number of characters in a TextInput text field. This property is read only.
`TextInput.maxChars`	The maximum number of characters that a user can enter in a TextInput text field.
`TextInput.maxHPosition`	The maximum possible value for TextField.hPosition. This property is read-only.
`TextInput.password`	A Boolean value that indicates whether or not the input 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 text field.

Inherits all methods from UIObject and UIComponent.

Event summary for the TextInput class

Event	Description
`TextInput.change`	Broadcast when the Input field changes.
`TextInput.enter`	Broadcast when the enter key is pressed.

Inherits all methods from UIObject and UIComponent.

TextInput.change

Availability

Flash Player 6 version 79.

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 prevent certain characters from being added to the component's text field; instead, use `TextInput.restrict`. This event is only triggered by user input, not by programmatic change.

The first usage example uses an `on()` handler and must be attached directly to a TextInput 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example sets a flag in the application that indicates if contents in the TextInput field have changed:

```
form.change = function(eventObj){
  // eventObj.target is the component which generated the change event,
  // i.e., the Input component.
  myFormChanged.visible = true;  // set a change indicator if the contents
  changed;
}
myInput.addEventListener("change", form);
```

See also

`UIEventDispatcher.addEventListener()`

TextInput.editable

Availability

Flash Player 6 version 79.

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 version 79.

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 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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

This example sets a flag in the application that indicates if contents in the TextInput field have changed:

```
form.enter = function(eventObj){
  // eventObj.target is the component which generated the enter event,
  // i.e., the Input component.
  myFormChanged.visible = true;
// set a change indicator if the user presses enter;
}
myInput.addEventListener("enter", form);
```

See also

`UIEventDispatcher.addEventListener()`

TextInput.hPosition

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

`textInputInstance``.hPosition`

Description

Property; defines the horizontal position of the text within the field. The default value is 0.

Example

The following code displays the leftmost characters in the field:

```
myTextInput.hPosition = 0;
```

TextInput.length

Availability

Flash Player 6 version 79.

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 version 79.

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; the `maxChars` property only indicates 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:

```
myTextInput.maxChars = 255;
```

TextInput.maxHPosition

Availability

Flash Player 6 version 79.

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 version 79.

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 the value of password is true, the text field is a password text field and hides the input characters. If 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 version 79.

Edition

Flash MX 2004.

Usage

textInputInstance.restrict

Description

Property; indicates the set of characters that a user may enter into the text field. The default value is undefined. If the value of the restrict property is null or empty string (""), a user can enter any character. If the value of the `restrict` property is a string of characters, you can enter only characters in the string into the text field; the string is scanned from left to right. A range may be specified using the dash (-).

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.

If the string begins with "^", all characters are initially accepted and succeeding characters in the string are excluded from the set of accepted characters. If the string does not begin with "^", no characters are initially accepted and succeeding characters in the string are included in the set of accepted characters.

The backslash character may be used to enter the characters "-", "^", and "\", as in the following:

```
\^
\-
\\
```

When you enter the `\` character in the Actions panel within `""` (double quotes), it has a special meaning for the Actions panel's double quotes interpreter. It signifies that the character following the `\` should be treated as is. For example, the following code is used 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 contained within double quotes, 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 quotes, 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 quotes. To send the value `0-9\-\^\\` to the .restrict interpreter, you must enter the following code:

```
myText.restrict = "0-9\\-\\^\\\\";
```

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 " ", the second "\" tells the interpreter that the next character should not be treated as a special character, as in the following:

```
myText.restrict = "0-9\\-\\^\\\\";
```

TextInput.text

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

textInputInstance.text

Description

Property; the text contents of a TextInput component. The default value is "" (empty string).

Example

The following code places a string in the myTextInput instance 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 will be 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 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
  }
}
```

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

```
class itemClass implements mx.data.to.TransferObject {
  function clone():Object {
    var b:ContactClass = new ContactClass();
    for (var p in this) {
      b[p] = this[p];
    }
    return b;
  }
}
```

TransferObject.getPropertyData()

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

```
class itemClass implements mx.data.to.TransferObject {
  function getPropertyData() {
  // your code here
  }
}
```

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

```
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

```
class Contact implements mx.data.to.TransferObject {

  function setPropertyData( data: Object ):Void {
    _readOnly = data.readOnly;
    phone = data.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 within 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 a disclosure triangle that a user can open to expose children. The children of a branch can either be leaves or branches themselves.

The data of a tree component must be provided from an XML data source. For more information, see the next section.

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.
Up arrow	Moves selection up one.
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 while authoring in Flash. The best way to create XML for the tree is to use the TreeDataProvider interface (Flash Professional only). 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 is composed of two sets of APIs: the Tree class and the TreeDataProvider interface. The Tree class contains the visual configuration methods and properties. The TreeDataProvider interface allows you to construct XML and add it to multiple tree instances. A TreeDataProvider object broadcasts changes to any trees that use it. As well, 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 548.

Formatting XML for the Tree component

The Tree component is designed to display hierarchical data structures. XML is the data model for the Tree component. 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 the `Tree.setIsBranch()` method.

Nodes in the XML data source can have any name. Notice in the sample above that each node is named with the generic name "node". The tree reads through the XML and builds the display hierarchy based on 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 a disclosure triangle 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; root nodes appear at the top level of the tree and have no parent. The icons are customizable; for more information, see "Using skins with the Tree component" on page 535.

There are many ways to structure XML. The Tree component is not designed to use all types of XML structures, so it's important to use XML that the Tree component can interpret. 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 allows the tree to know what it expects to render, and allows you to 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 `firstChild` (in the sample, `<node label="Mail">`), is rendered as the root node in the Tree view. This means that a tree can have multiple root folders. In this sample, 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.

Tree parameters

The following are authoring parameters that you can set for each Tree component instance in the Property inspector or in the Component Inspector panel:

multipleSelection A Boolean value that indicates whether a user can select multiple items (`true`) or not (`false`). The default value is false.

rowHeight 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 535.

You cannot enter data parameters in the Property inspector or in the Component Inspector panel for the Tree component like you can with other components. For more information, see "Using the Tree component (Flash Professional only)" on page 530 and "Creating an application with the Tree component" on page 532.

Creating an application with the Tree component

In this example, a developer is creating an e-mail application and chooses to use a Tree component to display the mailboxes.

You cannot enter data parameters in the Property inspector or in the Component Inspector panel like you can with other components. Because the data structure is more complex for Tree components, 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, 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:

1 In Flash, select File > New and select Flash Document.

2 In the Components panel, double-click the Tree component to add it to the Stage.

3 In the Property inspector, enter the instance name **myTree**.

4 In the Actions panel on Frame 1, enter the following code that creates a `change` event handler:

```
listenerObject = new Object();
listenerObject.change = function(evtObject){
  trace(evtObject.target.selectedItem.attributes.label + " was selected");
}
myTree.addEventListener("change", listenerObject);
```

The `trace` action inside the handler sends a message to the Output panel every time an item in the tree is selected.

5 Complete one of the following procedures to load or create an XML data source for the tree.

To load XML from an external file, do the following:

1 Follow the steps above to add a Tree component to an application and create a `change` event handler.

2 In the Actions panel on Frame 1, enter the following code:

```
myTreeDP = new XML();
myTreeDP.ignoreWhite = true;
myTreeDP.load("http://myServer.myDomain.com/source.xml");
myTreeDP.onLoad = function(){
  myTree.dataProvider = myTreeDP;
}
```

This code creates an ActionScript 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 ActionScript Dictionary Help.

3 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` actions 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, do the following:

1 Follow the steps in the first procedure above to add a Tree component to an application and create a `change` event handler.

2 In the Actions panel on Frame 1, enter the following code:

```
var myTreeDP = new XML();
myTreeDP.addTreeNode("Local Folders", 0);

// Use XML.firstChild to nest child nodes below Local Folders
var myTreeNode = 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 with the myTree component
myTree.dataProvider = myTreeDP;

// Set each of the 4 child nodes to be branches
for(var i=0; i<myTreeNode.childNodes.length; i++){
    var node = 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 API. 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.

3 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` actions in the `change` event handler send the data values to the Output panel.

To use the ActionScript XML class to create XML, do the following:

1 Follow the steps in the first procedure above to add a Tree component to an application and create a `change` event handler.

2 In the Actions panel on Frame 1, enter the following code:

```
// Create an XML object
var myTreeDP = new XML();
// Create node values
var myNode0 = myTreeDP.createElement("node");
myNode0.attributes.label = "Local Folders";
myNode0.attributes.data = 0;

var myNode1 = myTreeDP.createElement("node");
myNode1.attributes.label = "Inbox";
myNode1.attributes.data = 1;

var myNode2 = myTreeDP.createElement("node");
myNode2.attributes.label = "Outbox";
myNode2.attributes.data = 2;

var myNode3 = myTreeDP.createElement("node");
myNode3.attributes.label = "Sent Items";
myNode3.attributes.data = 3;

var myNode4 = 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 with the Tree control
myTree.dataProvider = myTreeDP;
```

Please read the comments in the code (lines that begin with `//`) for a description of the code. For more information about the XML object, see its entry in ActionScript Dictionary Help.

3 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` actions 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, do the following:

1 Follow the steps in the first procedure above to add a Tree component to an application and create a `change` event handler.

2 In the Actions panel on Frame 1, enter the following code:

```
myTreeDP = new XML("<node label='Local Folders'><node label='Inbox'
  data='0'/><node label='Outbox' data='1'/></node>");
myTree.dataProvider = myTreeDP;
```

The code above creates an XML object `myTreeDP` and assigns it to the `dataProvider` property of `myTree`.

3 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` actions 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 both 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 clips.

Using styles with the Tree component

For the latest information about this feature, click the Update button at the top of the Help tab.

Using skins with the Tree component

For the latest information about this feature, click the Update button at the top of the Help tab.

Tree class (Flash Professional only)

Inheritance UIObject > UIComponent > View > ScrollView > ScrollSelectList > List > Tree

ActionScript Class Name mx.controls.Tree

Method summary for 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 branch is open or closed.
`Tree.getNodeDisplayedAt()`	Returns the display index of a given node.
`Tree.getTreeNodeAt()`	Returns a node on the root of 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.setIsBranch()`	Indicates whether a node is a branch (receives folder icon and expander arrow).
`Tree.setIcon()`	Specifies whether a node is open or closed.
`Tree.setIsOpen()`	Specifies a symbol to be used as an icon for a node.

Inherits all methods from UIComponent, UIObject, View, ScrollView, ScrollSelectList, and List.

Property summary for 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.

Inherits all properties from UIComponent, UIObject, View, ScrollView, ScrollSelectList, and List.

Event summary for 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.

Inherits all events from UIComponent, UIObject, View, ScrollView, ScrollSelectList, and List.

Tree.addTreeNode()

Availability

Flash Player 6 version 79.

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 either constructed 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.

Note: 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 version 79.

Edition

Flash MX Professional 2004.

Usage

Usage 1:

```
myTree.addTreeNodeAt(index, label [, data])
```

Usage 2:

```
myTree.addTreeNodeAt(index, child)
```

Parameters

index The order (among the child nodes) in 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 either constructed 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.

Note: 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myTree.dataProvider
```

Description

Property; the `dataProvider` property can be either XML or a string. If the `dataProvider` is an XML object, it is added directly to the tree. If the `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 objects.

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 and Flash does not ignore that white space unless you set the `ignoreWhite` property to `true`.

Tree.firstVisibleNode

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myTree.firstVisibleNode
```

Description

Property; the first node at the top of the display. If the node is under 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. This value of this property is an XMLNode object.

Note: Setting this property is analogous to setting `List.vPosition`.

Example

The following example sets the scroll position to the top of the display:

```
myTree.firstVisibleNode = myTree.getTreeNodeAt(0);
```

Tree.getIsBranch()

Availability

Flash Player 6 version 79.

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 (is a *branch*). This is set automatically when children are added to the node. You only need to call `setIsBranch()` to create empty folders. For more information, see `Tree.setIsBranch()`.

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 version 79.

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 not (`false`).

Description

Method; indicates whether the specified node is open or closed.

Example

The following code assigns the state of the node to a variable:

```
var open = myTree.getIsOpen(myTree.getTreeNodeAt(1));
```

Tree.getDisplayIndex()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myTree.getDisplayIndex(node)
```

Parameters

node An XMLNode object.

Returns

The index of the node specified, 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 is an array of items that can be viewed in the tree window. For example, any children of a closed not 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.

Example

The following code gets the display index of `myNode`:

```
var x = myTree.getDisplayIndex(myNode);
```

Tree.getNodeDisplayedAt()

Availability

Flash Player 6 version 79.

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 checking `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 not 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myTree.getTreeNodeAt(index)
```

Parameters

index The index number of a tree.

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 version 79.

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.

V2 components use a dispatcher/listener event model. The Tree component broadcasts a `nodeClose` event when one of its nodes is clicked closed and 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 a set of 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 about event objects, see "Event Objects" on page 562.

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, as follows:

```
myTreeListener = new Object();
myTreeListener.nodeClose = function(evtObject){
  trace(evtObject.node + " node was closed");
}
myTree.addEventListener("nodeClose", myTreeListener);
```

Tree.nodeOpen

Availability

Flash Player 6 version 79.

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.

V2 components use a dispatcher/listener event model. The Tree component dispatches a `nodeOpen` event when a node is clicked open by a user 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 a set of 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 about event objects, see "Event Objects" on page 562.

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 `nodeOpen` handler in the `evtObject` parameter. When the `nodeOpen` event is broadcast, a `trace` statement is sent to the Output panel, as follows:

```
myTreeListener = new Object();
myTreeListener.nodeOpen = function(evtObject){
  trace(evtObject.node + " node was opened");
}
myTree.addEventListener("nodeOpen", myTreeListener);
```

Tree.removeAll()

Availability

Flash Player 6 version 79.

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 version 79.

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 that is was created.

Returns

An XMLNode object, or undefined if there is an error.

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.setIsBranch()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myTree.setIsBranch(node, isBranch)
```

Parameters

`node` An XML node.

`isBranch` A Boolean value indicating whether the node is a branch (`true`), or not (`false`).

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 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 the `setIsBranch()` method refreshes any views.

Example

The following code makes a node of `myTree` a branch;

```
myTree.setIsBranch(myTree.getTreeNodeAt(1), true);
```

Tree.setIcon()

Availability

Flash Player 6 version 79.

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 The linkage identifier of a symbol to be used as an icon beside the node. This parameter is used for the icon that represents the open state of branch nodes.

Returns

Nothing.

Description

Method; specifies an icon for the specified node. This method takes one parameter (*linkID*) for leaf nodes and two parameters (*linkID* and *linkID2*) for branches (the closed and open icons). The second parameter is ignored for leaf (non-branch) nodes, and if only one parameter is specified for a branch node, 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.setIsOpen()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myTree.setIsOpen(node, isOpen[, noEvent])
```

Parameters

node An XML node.

isOpen A Boolean value that opens a node (`true`) or closes it (`false`).

noEvent A Boolean value that animates the opening transition (`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);
```

Tree.selectedNode

Availability

Flash Player 6 version 79.

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 version 79.

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`

TreeDataProvider interface (Flash Professional only)

The TreeDataProvider is an interface; it does not need to be instantiated to be used. If a Tree class is packaged in a SWF, all XML instances in the SWF contain the TreeDataProvider API. All nodes in a Tree are XML objects that contain the TreeDataProvider API.

It's best to use the TreeDataProvider API methods to create XML for the `Tree.dataProvider` property because only TreeDataProvider broadcasts events to Tree components that refresh the tree's display. Built-in XML class methods can be used to create XML, but they don't broadcast events that will refresh the display.

You can use the TreeDataProvider API methods to control the data model and the data display. You can use built-in XML class methods for read-only tasks like traversing through the tree hierarchy.

The property that holds the text to be displayed can be selected by specifying a `labelField` or a `labelFunction` property. For example, the `code myTree.labelField = "fred";` results in the value of the property `myTreeDP.attributes.fred` being queried for the display text.

Method summary for the TreeDataProvider interface

Event	Description
`TreeDataProvider.addTreeNode()`	Adds a child node at the end of a parent node.
`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 descendents from the node's parent.
`TreeDataProvider.removeTreeNodeAt()`	Removes a node and all the node's descendents from the index position of the child node.

Property summary for 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 version 79.

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 either constructed 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 instance.

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, as follows:

```
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 version 79.

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 to 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 either constructed 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 instance.

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 version 79.

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 within the XML node 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 version 79.

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 XML node object. Setting this property does not refresh the displays of any tree.

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 version 79.

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 gets the second child of `myTreeNode`:

```
var myTreeNode = myTreeDP.firstChild.firstChild;
myTreeNode.getTreeNodeAt(1);
```

TreeDataProvider.removeTreeNode()

Availability

Flash Player 6 version 79.

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 descendents, from its parent.

Example

The following code removes a node:

```
myTreeDP.firstChild.removeTreeNode();
```

TreeDataProvider.removeTreeNodeAt()

Availability

Flash Player 6 version 79.

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 descendents) 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 a given node:

```
myTreeDP.firstChild.removeTreeNodeAt(3);
```

UIComponent

Inheritance UIObject > UIComponent

ActionScript Class Name mx.core.UIComponent

All v2 components extend UIComponent; it is not a visual component. The UIComponent class contains functions and properties that allow Macromedia components to share some common behavior. The UIComponent class allows you to do the following:

- Receive focus and keyboard input
- Enable and disable components
- Resize by layout

To use the methods and properties of the UIComponent, you call them directly from whichever component you are using. For example, to call the `UIComponent.setFocus()` method 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 the Macromedia Component V2 Architecture to create a new component. Even in that case, UIComponent is often created implicitly by other subclasses like Button. If you do need to create an instance of UIComponent, use the following code:

```
class MyComponent extends UIComponent;
```

Method summary for the UIComponent class

Method	Description
`UIComponent.getFocus()`	Returns a reference to the object that has focus.
`UIComponent.setFocus()`	Sets focus to the component instance.

Inherits all methods from the UIObject class.

Property summary for 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.

Inherits all properties from the UIObject class.

Event summary for 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.

Inherits all events from the UIObject class.

UIComponent.focusIn

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

```
on(focusIn){
  ...
}
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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

The following code disables a button while a user types in the text field txt:

```
txtListener.handleEvent = function(eventObj) {
  form.button.enabled = false;
}
txt.addEventListener("focusIn", txtListener);
```

See also

UIEventDispatcher.addEventListener()

UIComponent.focusOut

Availability

Flash Player 6 version 79.

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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

The following code enables a button when a user leaves the text field `txt`:

```
txtListener.handleEvent = function(eventObj){
  if (eventObj.type == focusOut){
      form.button.enabled = true;
  }
}
txt.addEventListener("focusOut", txtListener);
```

See also

`UIEventDispatcher.addEventListener()`

UIComponent.enabled

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

componentInstance.enabled

Description

Property; indicates whether the component can accept focus and mouse input. If the value is `true`, it can receive focus and input; if the value is `false`, it can't. The default value is `true`.

Example

The following example sets the `enabled` property of a CheckBox component to `false`:

```
checkBoxInstance.enabled = false;
```

UIComponent.getFocus()

Availability

Flash Player 6 version 79.

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 version 79.

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 should not be used unless necessary because it can impact 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 very low-level event that should not be used unless necessary because it can impact 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 sets focus to the `checkbox` instance:

```
checkbox.setFocus();
```

UIComponent.tabIndex

Availability

Flash Player 6 version 79.

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

ActionScript Class Name mx.events.EventDispatcher; mx.events.UIEventDispatcher

Events allow you to know when the user has interacted with a component, and also to know when important changes have happened in the appearance or life cycle of a component, such as the creation or destruction of a component or if its size changes.

Each component broadcasts different events and those events are listed in each component entry. There are several ways to use component events in ActionScript code. For more information, see "About component events" on page 22.

Use the `UIEventDispatcher.addEventListener()` to register a listener with a component instance. The listener is invoked when a component's event is triggered.

UIEventDispatcher.addEventListener()

Availability

Flash Player 6 version 79.

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 is triggered, 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 is triggered. 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 is triggered. For more information, see "Using component event listeners" on page 22.

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:

```
lo = new Object();
lo.click = function(evt){
  if (evt.target == button1){
    trace("button 1 clicked");
  } else if (evt.target == button2){
    trace("button 2 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 within the `if` statements to determine which button instance was clicked. For more information, see "Event Objects" on page 562.

Example

The following example defines a listener object, `myListener`, and defines the callback function click. It then calls `addEventListener()` to register the `myListener` listener object with the component instance `myButton`. To test this code, place a button component on the Stage with the instance name `myButton`, and place the following code in Frame 1:

```
myListener = new Object();
myListener.click = function(evt){
  trace(evt.type + " triggered");
}
myButton.addEventListener("click", myListener);
```

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 but aren't implemented in version 2 (v2) of the Macromedia Component Architecture. Every v2 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.

Properties of the event object

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

UIObject

Inheritance MovieClip > UIObject

ActionScript Class Name mx.core.UIObject

UIObject is the base class for all v2 components; it is not a visual component. The UIObject class wraps the ActionScript MovieClip object and contains functions and properties that allow Macromedia v2 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 the API 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, 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 the Macromedia Component v2 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

Method	Description
`UIObject.createObject()`	Creates a subobject on an object.
`UIObject.createClassObject()`	Creates an object on the specified class.
`UIObject.destroyObject()`	Destroys a component instance.
`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 draws in the current frame.
`UIObject.setSize()`	Resizes the object to the requested size.
`UIObject.setSkin()`	Sets a skin in the object.

Property summary for 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 position 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 position of the object in pixels. Read-only.
`UIObject.y`	Returns the position of the top edge of the object relative to its parent. Read-only.

Event summary for 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 the subobjects are being unloaded.
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 version 79.

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 the UIObject.move() method.

Example

This example moves the check box so it aligns under the bottom edge of the listbox:

```
myCheckbox.move(myCheckbox.x, form.height - listbox.bottom);
```

UIObject.createObject()

Availability

Flash Player 6 version 79.

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 panel.

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 that is an instance of the symbol.

Description

Method; creates a subobject on an object. Generally only used by component or advanced developers.

Example

The following example creates a CheckBox instance on the `form` object:

```
form.createObject("CheckBox", "sym1", 0);
```

UIObject.createClassObject()

Availability

Flash Player 6 version 79.

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 that is an instance of the specified class.

Description

Method; creates a subobject of an object. Generally only used by component or advanced developers. This method allows you to create components at runtime.

You need to specify the class package name. Do one of the following:

```
import mx.controls.Button;
createClassObject(Button,"button2",5,{label:"Test Button"});
```

or

```
createClassObject(mx.controls.Button,"button2",5,{label:"Test Button"});
```

Example

The following example creates a CheckBox object:

```
form.createClassObject(CheckBox, "cb", 0, {label:"Check this"});
```

UIObject.destroyObject()

Availability

Flash Player 6 version 79.

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.draw

Availability

Flash Player 6 version 79.

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 very low-level event that should not be used 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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

`UIEventDispatcher.addEventListener()`

UIObject.height

Availability

Flash Player 6 version 79.

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 the `UIObject.setSize()` property.

Example

The following example makes the check box taller:

```
myCheckbox.setSize(myCheckbox.width, myCheckbox.height + 10);
```

UIObject.hide

Availability

Flash Player 6 version 79.

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.getStyle()

Availability

Flash Player 6 version 79.

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 styleDeclaration or object. If the style property is an inheriting style, the parents of the object may be the source of the style value.

For a list of the styles supported by each component, see their individual entries.

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.invalidate()

Availability

Flash Player 6 version 79.

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.

Example

The following example marks the ProgressBar instance `pBar` for redraw:

```
pBar.invalidate();
```

UIObject.left

Availability

Flash Player 6 version 79.

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 the `UIObject.move()` method.

UIObject.load

Availability

Flash Player 6 version 79.

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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 only pass integral values to the `UIObject.move()` or the component may appear fuzzy.

Example

This example move the checkbox to the right 10 pixels:

```
myCheckbox.move(myCheckbox.x + 10, myCheckbox.y);
```

UIObject.redraw()

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

```
componentInstance.redraw(always)
```

Parameters

always A Boolean value. If `true`, draws the object even if `invalidate()` wasn't called. If `false`, draws the object only if `invalidate()` was called.

Returns

Nothing.

Description

Method; forces validation of the object so it draws 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 version 79.

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 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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 version 79.

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 version 79.

Edition

Flash MX 2004.

Usage

```
componentInstance.right
```

Description

Property (read-only); a number indicating the right position of the object in pixels relative to its parent's right side. To set this property, call the `UIObject.move()` method.

Example

The following example moves the check box so it aligns under the right edge of the listbox:

```
myCheckbox.move(form.width - listbox.right, myCheckbox.y);
```

UIObject.scaleX

Availability

Flash Player 6 version 79.

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 version 79.

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 version 79.

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 only pass integral values to the `UIObject.setSize()` or the component may appear fuzzy. This method (and all methods and properties of UIObject) is available from any component instance.

When you call this method on an instance of the ComboBox, 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 version 79.

Edition

Flash MX 2004.

Usage

```
componentInstance.setSkin(id, linkageName)
```

Parameters

id A number indicating the variable. This value is usually a constant defined in the class definition.

linkageName A string indicating an asset in the library.

Returns

Nothing.

Description

Method; sets a skin in the component instance. Use this method when you are developing components. You cannot use this method to set a component's skins at runtime.

Example

This example sets a skin in the `checkbox` instance:

```
checkbox.setSkin(CheckBox.skinIDCheckMark, "MyCustomCheckMark");
```

UIObject.setStyle()

Availability

Flash Player 6 version 79.

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.

Returns

A UIObject 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 their individual entries.

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 version 79.

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 the `UIObject.move()` method.

UIObject.unload

Availability

Flash Player 6 version 79.

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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

Example

The following example deletes `sym1` when the `unload` event is triggered:

```
formListener.handleEvent = function(eventObj){
  // eventObj.target is the component which generated the change event,
  form.destroyObject(sym1);
}
form.addEventListener("unload", formListener);
```

UIObject.visible

Availability

Flash Player 6 version 79.

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 version 79.

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 the `UIObject.setSize()` method.

Example

The following example makes the check box wider:

```
myCheckbox.setSize(myCheckbox.width + 10, myCheckbox.height);
```

UIObject.x

Availability

Flash Player 6 version 79.

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 the `UIObject.move()` method.

Example

The following example moves the check box to the right 10 pixels:

```
myCheckbox.move(myCheckbox.x + 10, myCheckbox.y);
```

UIObject.y

Availability

Flash Player 6 version 79.

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 the `UIObject.move()` method.

Example

The following example moves the check box down 10 pixels:

```
myCheckbox.move(myCheckbox.x, myCheckbox.y + 10);
```

Web service classes (Flash Professional only)

The classes found in the mx.services package consist of classes for accessing web services that use Simle Object Access Protocol (SOAP). This WebService API is not the same as the WebServiceConnector component API. The former is a set of classes that can you use only in ActionScript code, and is common with various Macromedia products. The latter is an API unique to Flash MX 2004, and provides an ActionScript interface to the visual authoring tool for the WebServiceConnector component.

Making web service classes available at runtime (Flash Professional only)

In order to make the web service classes available at runtime, the WebServiceClasses 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, see "Working with data binding and web services at runtime (Flash Professional only)" in Using Flash Help.

Note: These classes are automatically made available to your Flash document when you add a WebServiceConnector component to your FLA.

Classes in the mx.services package (Flash Professional only)

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 the classes are listed in the table.

Class	Description
WebService class (Flash Professional only)	Using a 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 results and faults of the call.
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.

Log class (Flash Professional only)

The Log class is part of the mx.services package and is intended to be used with the WebService class (see "WebService class (Flash Professional only)" on page 596). For an overview of the classes in the mx.data.services package, see "Web service classes (Flash Professional only)" on page 581.

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, such as sending the log messages to a trace command.

The constructor for this object creates a Log object that can be passed as an optional argument to the WebService constructor (see "WebService class (Flash Professional only)" on page 596).

ActionScript Class Name mx.services.Log

Callback summary for the Log object

Callback	Description
`Log.onLog()`	Sends a log message to a log object.

Constructor for the Log class

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

myWebSrvcLog = new Log([logLevel] [, logName]);

Parameters

`logLevel` Log level to indicate the types of information you want to record in the log. In the web services code, the log messages are broken down into categories or levels. The `logLevel` parameter of the Log object constructor relates to these categories. Three `logLevels` are available:

- `Log.BRIEF`: The log records primary life-cycle event and error notifications.
- `Log.VERBOSE`: The log records all life-cycle event and error notifications.
- `Log.DEBUG`: The log records metrics and fine-grained events and errors.

The default `logLevel` is `log.BRIEF`.

`logName` Optional name that is included with each log message. If you are using multiple log objects, you can use the logName to determine which log recorded a given message.

Returns

Nothing.

Description

Constructor; creates a Log object. Use this constructor to create a log. After you create the Log object, you can pass this object to a web service to get messages.

Example

You can call on the `new Log` constructor which returns 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myWebSrvcLog.onLog = function(message)
```

Parameters

message The log message passed to the handler. For more information about log messages, see "Log class (Flash Professional only)" on page 581.

Returns

None.

Description

Log callback function; Flash Player calls this function when a log message is sent to a log file. This handler is a good place to put code that records or displays the log messages, such as a `trace` command. The Log construction is described in "Log class (Flash Professional only)" on page 581.

Example

The following example creates a new log object, passes it to a new WebService object and handles the logging messages.

```
// creates a new log object
myWebSrvcLog = new Log();
// passes the log object to the web service
myWebService = new WebService(wsdlURI, myWebSrvcLog);
// handles in-coming log messages
myWebSrvcLog.onLog = function(message)
{
  mytrace("Log Event:\r myWebSrvcLog.message="+message+);
}
```

PendingCall class (Flash Professional only)

The PendingCall class is part of the mx.services package and is intended to be used with the WebService class (see "WebService class (Flash Professional only)" on page 596). For an overview of the classes in the mx.data.services package, see "Web service classes (Flash Professional only)" on page 581.

When you call a method on a WebService object, the WebService object returns a PendingCall object. The PendingCall object is not constructed by the developer. You use the `onResult` and `onFault` callbacks of the PendingCall object to handle the asynchronous response from the web service method. If the web service method returns a fault, Flash Player calls the `PendingCall.onFault` callback and passes a SOAPFault object that represents the XML SOAP fault returned by the server/web service. If the web service invocation is successful, Flash Player calls the `PendingCall.onResult` callback 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 596.

The PendingCall object also offers you access to output parameters when there are more than one. Many web services return only a single result, but some web services return more than one result. The "return value" referred to in this API is simply the first (or only) result. The `PendingCall.getOuputXXX` functions give you access to all of the results, not just the first. So while the "return value" is handed to you in the argument to the `onResult()` callback, if there are other output parameters you want to access, use `getOutputValues()` (returns an Array) and `getOutputValue(index)` (returns an individual one) 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` contains the ActionScript value of an output parameter, and `element` contains the XML value of the output parameter. The following functions return a SOAPParameter object, or an array of SOAPParameter objects, which contains the value (`param.value`) as well as the XML element (`param.element`): `getOutputParameters()`, `getOutputParameterByName(name)`, and `getOutputParameter(index)`.

ActionScript Class Name mx.services.PendingCall

Function summary for the PendingCall object

Function	Description
`PendingCall.getOutputParameter()`	Gets a SOAPParameter object based on the `index` passed in.
`PendingCall.getOutputParameterByName()`	Gets a SOAPParameter object based on the `localName` passed in.
`PendingCall.getOutputParameters()`	Gets an array of SOAPParameter objects.
`PendingCall.getOutputValue()`	Gets the output value based on the index passed in.
`PendingCall.getOutputValues()`	Gets an array of all the output values.

Property summary for the PendingCall object

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

Callback	Description
`PendingCall.onFault()`	Called by a web service when the method fails.
`PendingCall.onResult()`	Called when a method has succeeded and returned a result.

Constructor for the PendingCall class

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Description

The PendingCall object is not constructed by the developer. Instead, when you call a function on a `WebService` object, the `WebService` object returns a `PendingCall` object.

PendingCall.getOutputParameter()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myPendingCall.getOutputParameter(var index)
```

Parameters

index The index of the parameter.

Returns

SOAPParameter object with the following elements:

Element	Description
value	The ActionScript value of the parameter.
element	The raw XML of the parameter in the SOAP envelope.

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 handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use functions such as this one or `getOutputValue()`. The `getOutputParameter()` function returns the *`nth`* output parameter as a SOAPParameter object.

See also `PendingCall.getOutputValue()`, `PendingCall.getOutputValues()`, `PendingCall.getOutputParameterByName()` and `PendingCall.getOutputParameters()`.

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>
...
```

PendingCall.getOutputParameterByName()

Availability

Flash Player 6 version 79.

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

SOAPParameter object with the following elements:

Element	Description
value	The ActionScript value of the parameter.
element	The raw XML of the parameter in the SOAP envelope.

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 handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use APIs such as this one. The `getOutputParameterByName()` call returns the output parameter with the name *`localName`*.

See also `PendingCall.getOutputValue()`, `PendingCall.getOutputValues()`, `PendingCall.getOutputParameter()` and `PendingCall.getOutputParameters()`.

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>
...
```

PendingCall.getOutputParameters()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

`myPendingCall``.getOutputParameterByName()`

Parameters

None.

Returns

Array of SOAPParameter objects with the following elements:

Element	Description
value	The ActionScript value of the parameter.
element	The raw XML of the parameter in the SOAP envelope.

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 handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use APIs such as this one or `getOutputValues()`.

See also `PendingCall.getOutputValue()`, `PendingCall.getOutputValues()`, `PendingCall.getOutputParameterByName()` and `PendingCall.getOutputParameter()`.

PendingCall.getOutputValue()

Availability

Flash Player 6 version 79.

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 *nth* 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 handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use APIs such as this one or `getOutputParameter()`. The `getOutputValue()` call returns the *nth* output parameter.

See also `PendingCall.getOutputParameter()`, `PendingCall.getOutputValues()`, `PendingCall.getOutputParameterByName()` and `PendingCall.getOutputParameters()`.

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>
...
```

PendingCall.getOutputValues()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

myPendingCall.getOutputValues()

Parameters

None.

Returns

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 handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use APIs such as this one or `getOutputParameters()`.

See also `PendingCall.getOutputValue()`, `PendingCall.getOutputParameter()`, `PendingCall.getOutputParameterByName()` and `PendingCall.getOutputParameters()`.

PendingCall.myCall

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

PendingCall.myCall

Description

Property; the SOAPCall object corresponding to the PendingCall's 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 593.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myPendingCallObj.onFault = function(fault)
{
  // handles any faults, for example, by telling the
  // user that the server isn't available or to contact technical
  // support

}
```

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 `PendingCall.onfault()` is a SOAPFault object. It is not constructed directly by developers, but returned as the result of a failure. This object is an ActionScript mapping of the SOAP Fault 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

PendingCall object callback function; you provide this handler that Flash Player calls when a web service method has failed and returned an error. The fault parameter is an ActionScript SOAPFault object.

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myPendingCallObj.onResult = function(result)
{
  // catches the result and handles it for this application

}
```

Parameters

result Decoded ActionScript object version of the XML result returned by a web service method called with `myPendingCallObj = myWebService.methodName(params)`.

Returns

None.

Description

PendingCall callback function; you provide this handler that 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. To get the raw XML returned instead of the decoded result, access the *PendingCall*`.response` property (see PendingCall.response).

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;
}
```

PendingCall.request

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
rawXML = myPendingCallback.request;
```

Description

PendingCall 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 that gets sent over the wire. Use this property to access the raw XML of the request. Use myPendingCallback`.onResult()` to get the ActionScript version of the results of the request.

PendingCall.response

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
rawXML = myPendingCallback.response;
```

Description

PendingCall 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 that gets sent over the wire. Use myPendingCallback`.onResult()` to get the corresponding ActionScript version of the results of the request.

SOAPCall class (Flash Professional only)

The SOAPCall class is part of the mx.services package and is intended as an advanced feature to be used with the WebService class (see "WebService class (Flash Professional only)" on page 596). For an overview of the classes in the mx.data.services package, see "Web service classes (Flash Professional only)" on page 581.

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 is the descriptor of the operation, and as such contains all the information about that particular operation (how the XML should look on the wire, the operation style, and so on). It also provides control over certain behaviors. You can get the SOAPCall for a given operation by using the `getCall(operationName)` function. There is a single SOAPCall for each operation, shared by all active calls to that operation. Once you have the SOAPCall, you can customize the descriptor, by doing the following:

- Turn on/off decoding of the XML response.
- Turn on/off the delay of converting SOAP arrays into ActionScript objects.
- Change the concurrency configuration for a given operation.
- Add a header to the SOAPCall object.

ActionScript Class Name mx.services.SOAPCall

Function summary for the SOAPCall object

Function	Description
`SOAPCall.addHeader()`	Adds a header to the SOAPCall object.

Property summary for the SOAPCall object

Property	Description
`SOAPCall.concurrency`	Changes the concurrency configuration for a given operation.
`SOAPCall.doDecoding`	Turns on/off decoding of the XML response.
`SOAPCall.doLazyDecoding`	Turns on/off the delay of turning SOAP arrays into ActionScript objects.

Constructor for the SOAPCall class

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Description

The SOAPCall object is not constructed by the developer. 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`.

SOAPCall.addHeader()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
SOAPCall.addHeader(var header)
```

Parameters

header Header to be added.

Returns

None.

Description

Function; adds a header to the SOAPCall object.

Example

The following example creates a new SOAP header and attaches it to the SOAPCall. The following code:

```
import mx.services.QName;

var qname = new QName("bar", "http://foo");
var value = "hi there!";
var header = new SOAPHeader(qname, value);
soapCall.addHeader(header);
```

creates the following SOAP header:

```
...
<SOAP:Header>
  <ns1:bar
    xmlns:ns1="http://foo"
    xsi:type="xsd:string">hi there!</ns1:bar>
</SOAP:Header>
...
```

SOAPCall.concurrency

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
SOAPCall.concurrency
```

Description

Property; number of concurrent requests. Possible values are listed in the table below:

Value	Description
`SOAPCall.MULTIPLE_CONCURRENCY`	Allow multiple active calls.
`SOAPCall.SINGLE_CONCURRENCY`	Allow only one call at a time by faulting after one is active.
`SOAPCall.LAST_CONCURRENCY`	Allow only one call by cancelling previous ones.

SOAPCall.doDecoding

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

`SOAPCall``.doDecoding`

Description

Property; turns on/off decoding of the XML response—by default the XML response is converted (decoded) into ActionScript objects. If you just want the XML, you can set `SOAPCall.doDecoding = false`.

SOAPCall.doLazyDecoding

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

`SOAPCall``.doLazyDecoding`

Description

Property; turns on/off "lazy decoding" of arrays. By default we use a "lazy" decoding algorithm to delay turning SOAP arrays into ActionScript objects until the last moment—this makes operations return a lot faster when large data sets are returned. This means any arrays you get back from the remote end are ArrayProxy objects. Then when you access a particular index (`foo[5]`) that element is automatically decoded if necessary. This behavior can be turned off (which will cause all arrays to be fully decoded) by setting `SOAPCall.doLazyDecoding = false`.

WebService class (Flash Professional only)

The WebService class is part of the mx.services package and is intended to be used with the following classes:

- Log class (Flash Professional only)
- PendingCall class (Flash Professional only)
- SOAPCall class (Flash Professional only)

Note: This WebService API is not the same as the WebServiceConnector component API. The former is a set of classes that can you use only in ActionScript code, and is common with various Macromedia products. The latter is an API unique to Flash MX 2004, and provides an ActionScript interface to the visual authoring tool for the WebServiceConnector component.

For an overview of the classes in the mx.services package, see "Web service classes (Flash Professional only)" on page 581.

The WebServices 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 `onLoad()` callback is invoked. If there is a problem loading the WSDL, the `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 method invocations is `PendingCall`. These objects are normally not constructed by developers, but instead are constructed automatically as a result of the `webServiceObject.`*`webServiceMethodName`*`()` command. These objects are not the result of the WebService call, which comes later. Instead, the PendingCall object represents the call in progress. When the WebService operation completes (usually several seconds after a method call is made), the various PendingCall data fields are filled in, and the `onResult` or `onFault` callback you provide is called. For more information about the PendingCall object, see "PendingCall class (Flash Professional only)" on page 584.

The Player queues up 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 to correctly encode and send a SOAP request. Function calls that you make after the WSDL has been parsed do not need to be queued; they happen 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.

ActionScript Class Name mx.services.WebService

Using the WebServices API

The WebServices API, included under the mx.services package, consists of the WebService class, the Log class, the PendingCall class, and the PendingCall class.

Supported types

The WebService feature supports a subset of XML Schema types as defined in the tables below.

Complex 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

XML Schema type	ActionScript Binding
decimal	Number
integer	Number
negativeInteger	Number
nonNegativeInteger	Number
positiveInteger	Number
long	Number
int	Number
short	Number
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

```
schema
     complexType
          complexContent
               restriction
          sequence | simpleContent
               restriction
     element
          complexType | simpleType
```

WebService security

The WebService API conforms 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 for the user 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.

Function summary for the WebService object

Function	Description
`WebService.myMethodName()`	Invokes a specific web service operation defined by the WSDL.
`WebService.getCall()`	Gets the SOAPCall for a given operation

Callback summary for the WebService object

Callback	Description
`WebService.onLoad()`	Called when the web service has successfully loaded and parsed its WSDL file.
`WebService.onFault()`	Called when an error occurred during WSDL parsing.

Constructor for the WebService class

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myWebServiceObject = new WebService(wsdlURI [, logObject]);
```

Parameters

The constructor parameters are as follows:

`wsdlURI` URL of the web service WSDL file.

`logObject` Optional parameter specifying the name of the Log object for this web service (see "Log class (Flash Professional only)" on page 581).

Returns

None.

Description

To create a WebService object, you call `new WebService()` and provide a WSDL URL. Flash Player returns a WebService object. The WebService object constructor can optionally accept a Log object and a proxy URL:

```
myWebServiceObject = new WebService(wsdlURI [, logObject]);
```

If you want to, you can utilize two callbacks for the WebService object. Flash Player calls the `webServiceObject.onLoad(WSDLDocument)` function 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 the `webServiceObject.onFault(fault)` when an error occurs in finding or parsing the WSDL file. This is a good place to put debugging code and code that tells the user 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)`, then invoke the method in the following manner:

```
myPendingCallObject = myWebServiceObject.getCompanyInfo(tickerSymbol);
```

In the previous example, the callback object is named `myPendingCallObject`. All method invocations are asynchronous, and return a callback object of type `PendingCal`. Asynchronous means that the results of the web service call are not available immediately.

When you make the call

```
x = stockService.getQuote("macr");
```

the object `x` is not the results of `getQuote` (it's a PendingCall object). The actual results are only available later on (usually several seconds 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 584). For 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
getCall(var operationName)
```

Parameters

operationName The web service operation of the corresponding SOAPCall that you want to retrieve.

Returns

SOAPCall object.

Description

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 is the descriptor of the operation, and as such contains all the information about that particular operation (how the XML should look on the wire, the operation style, and so on). It also provides control over certain behaviors. You can get the SOAPCall for a given operation by using the `getCall(operationName)` method. There is a single SOAPCall for each operation, shared by all active calls to that operation. Once you have the SOAPCall, you can change the operator descriptor by using the SOAPCall API. For more information, see "SOAPCall class (Flash Professional only)" on page 593.

Example

For an example on using this call, see "SOAPCall class (Flash Professional only)" on page 593.

WebService.onFault()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
MyWebServiceObject.onFault
```

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. It is not constructed directly by developers, but returned as the result of a failure. This object is an ActionScript mapping of the SOAP Fault 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

WebService callback function; Flash Player calls this function when the new `webService(WSDLUrl)` method 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 a 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
myService.onLoad
```

Parameters

wsdlDocument WSDL XML document.

Returns

None.

Description

Webservice callback function; Flash Player calls this callback when the WebService object has successfully loaded and parsed its WSDL file. Operations can be invoked in an application before this event occurs, but when this happens they will be 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
}
```

WebService.*myMethodName*()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
callbackObj = myWebServiceObject.myMethodName(param1, ... paramN);
```

Parameters

Parameters required depend on the web service method being called.

Returns

callbackObj PendingCall object to which you can attach function for handling results and errors on the invocation. For more information, see "PendingCall class (Flash Professional only)" on page 584.

The callback invoked when the response comes back from the WebService method is `PendingCall.onResult()`, or `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

To invoke a web service operation, invoke it as a method directly available on the web service. For example, if your web service has the method `getCompanyInfo(tickerSymbol)`, then call:

```
myCallbackObject.myservice.getCompanyInfo(tickerSymbol);
```

All invocations are asynchronous, and return a callback object, of the object type `PendingCall`.

WebServiceConnector (Flash Professional only)

The WebServiceConnector component enables you to access remote methods exposed by a server using the industry-standard SOAP (Simple Object Access Protocol) protocol. A web service may accept parameters and return a result. Using the Flash MX Professional 2004 authoring tool and the WebServiceConnector component you can introspect, 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 a WebServiceConnector for each different operation you want to call.

A web service defines the methods (sometimes referred to as operations) that are available for consumption through an XML file using the Web Service Description Language (WSDL) format. The WSDL file specifies a list of operations, parameters and results (referred to as a schema) that are exposed by the web service.

WSDL files are accessible using a URL. In Flash MX Professional 2004, you can view the schema of any web service by entering the URL for its WSDL file using the Web Services panel. Once you identify a WSDL file, the web service is available to any application you create.

Only the WSDL file author can change the WSDL file or operation parameter. Whenever the author changes the WSDL file, the params and results schemas are updated. These changes will overwrite any edits the developer has made to the schema. To get an updated WSDL file, you can select Refresh Web Services from the Web Service panel menu.

The WebServiceConnector component and the XMLConnector component implement the RPC (Remote Procedure Call) Component API, a set of methods, properties, and events that define an easy way to send parameters to, and receive results from, an external data source.

A single instance of 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.

A developer can edit the schema to customize it for use in an application (for example, to provide additional formatting or validation settings). See "Working with schemas in the Schema tab (Flash Professional only)" in Using Flash Help.

Using the WebServiceConnector (Flash Professional only)

You can use the WebServiceConnector 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 web service URL for the web service. The WebServiceConnector appears on the Stage during application authoring, but has no visual appearance in the runtime application.

You can enter the URL for a web service in the Component Inspector panel or the Web Services panel. See "The WebServiceConnector component" in Using Flash Help.

For more information on working with the WebServiceConnector component, see "Data binding (Flash Professional only)" in Using Flash Help.

WebServiceConnector parameters

The following are authoring parameters that you can set for each WebServiceConnector component instance, in the Component Inspector panel Parameters tab:

`multipleSimultaneousAllowed` (Boolean type) indicates whether multiple calls can take place at the same time; the default value is false. If false, then the `trigger()` function will not perform a call if a call is already in progress. A status event will be emitted, with the code CallAlreadyInProgress. If true, then the call will take place.

`operation` (String type) is the name of an operation that appears within the SOAP port in a WSDL file.

`suppressInvalidCalls` (Boolean type) indicates whether to suppress a call if parameters are invalid; the default value is false. If true, then the `trigger()` function will not perform a call if the databound parameters fail the validation. A status event will be emitted, with the code InvalidParams. If false, then the call will take 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 panel. The service description is also added to the Web Service panel. For example, see www.xmethods.net/sd/2001/TemperatureService.wsdl.

WebServiceConnector class (Flash Professional only)

Inheritance RPC > WebServiceConnector

ActionScript Class Name mx.data.components.WebServiceConnector

Property summary for the WebServiceConnector class

Property	Description
WebServiceConnector.multipleSimultaneousAllowed	Indicates whether multiple calls can take place at the same time.
WebServiceConnector.multipleSimultaneousAllowed	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) within 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.

Method summary for the WebServiceConnector class

Method	Description
WebServiceConnector.trigger()	Initiates a remote procedure call.

Event summary for the WebServiceConnector class

Event	Description
WebServiceConnector.result	Broadcast when a call to a web service completes successfully.
WebServiceConnector.send	Broadcast when the `trigger()` function 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.multipleSimultaneousAllowed;
```

Description

Property; indicates whether multiple calls can take place at the same time. If false, then the `trigger()` function will perform a call if another call is already in progress. A `status` event will be emitted, with the code `CallAlreadyInProgress`. If true, then the call will take place.

When multiple calls are simultaneously in progress, there is no guarantee that they will complete in the same order as they were triggered. Also, Flash Player may place limits on the number of simultaneous network operations. This limit varies by version and platform.

Example

The following example enables multiple simultaneous calls to `myXmlUrl` take place:

```
myXmlUrl.multipleSimultaneousAllowed = true;
```

WebServiceConnector.operation

Availability

Flash Player 6 version 79.

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.

WebServiceConnector.params

Availability

Flash Player 6 version 79.

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, then the data type of `params` is an XML document of some kind.

If you use the Property Inspector or Component Inspector panel to set the WSDLURL and operation at during 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("result", myListenerObject);
```

Description

Event; broadcasts when a Remote Procedure Call operation successfully completes.

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);
```

WebServiceConnector.results

Availability

Flash Player 6 version 79.

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 storage will become available for garbage collection when the parent goes away.
- In ActionScript, you can assign null to this property at any time.

WebServiceConnector.send

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("send", myListenerObject);
```

Description

Event; broadcasts during the processing of a `trigger()` operation, 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 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("status", myListenerObject);
```

Description

Event; broadcasts 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 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 (a) the `trigger()` function is called, and (b) multipleSimultaneousAllowed is false, and (c) a call is already in progress. After this event occurs, the attempted call is considered complete, and there will be no "result" or "send" event.
InvalidParams	no data	This event is emitted if the `trigger()` function found that the "params" property did not contain valid data. If the "suppressInvalidCalls" property is true, then the attempted call is considered complete, and there will be no "result" or "send" event.

Here are the possible web service faults:

faultcode	faultstring	detail
Timeout	Timeout while calling method xxx	þ
MustUnderstand	No callback for header xxx	þ

faultcode	faultstring	detail
Server.Connection	Unable to connect to endpoint: xxx	þ
VersionMismatch	Request implements version: xxx Response implements version yyy	þ
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	þ
WSDL.UnrecognizedBindingName	The WSDL parser couldn't find a binding named xxx in namespace yyy	þ
WSDL.UnrecognizedPortTypeName	The WSDL parser couldn't find a portType named xxx in namespace yyy	þ
WSDL.UnrecognizedMessageName	The WSDL parser couldn't find a message named xxx in namespace yyy	þ
WSDL.BadElement	Element xxx not resolvable	þ
WSDL.BadType	Type xxx not resolvable	þ
Client.NoSuchMethod	Couldn't find method 'xxx' in service	þ
yyy	yyy - errors reported from server, this depends on which server you talk to	þ
No.WSDLURL.Defined	the WebServiceConnector component had no WSDL URL defined	þ
Unknown.Call.Failure	WebService invocation failed for unknown reasons	þ
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 version 79.

Edition

Flash MX Professional 2004.

Usage

componentInstance`.suppressInvalidCalls;`

Description

Property; indicates whether to suppress a call if parameters are invalid. If `true`, then the `trigger()` function will not perform a call if the bound parameters fail the validation. A "status" event will be emitted, with the code `InvalidParams`. If `false`, then the call will take place, using the invalid data as required.

WebServiceConnector.timeout

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

componentInstance`.timeout;`

Description

Property; a time period in seconds within which the web service connection will fail if results do not come back. A `status` event (inherited from the RPC component) is emitted, with the code `WebServiceFault`, faultcode `Timeout`.

WebServiceConnector.trigger()

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.trigger();
```

Description

Method; initiates a call to a web service. Each web servicedefines 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).

WebServiceConnector.WSDLURL

Availability

Flash Player 6 version 79.

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 during authoring, the WSDL file is immediately fetched and parsed. The resulting parameters and results appear in the Schema tab of the Component Inspector panel. The service description also appears in the Web Service panel.

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 non-modal. 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 panel while 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 only enable accessibility for a component once no matter how many instances you have of the component. For more information, see "Creating Accessible Content" in Using Flash Help.

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 PopUpManager to create modal windows that overlap other objects on the Stage. For more information, see Window class.

If you use the PopUpManager to add a Window component to a document, the Window instance will have its own FocusManager, distinct from the rest of the document. If you don't use the PopUpManager, the window's contents participate focus ordering with the other components in the document. For more information about controlling focus, see "Creating custom focus navigation" on page 24 or "FocusManager class" on page 270.

Window component parameters

The following are authoring parameters that you can set for each Window component instance in the Property inspector or in the Component Inspector panel:

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 JPG file to load into the window. The default value is "". Loaded content clips 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`.

You can write ActionScript to control these and additional options for Window components using its properties, methods, and events. For more information, see Window class.

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, do the following:

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 Options menu.

3 Check 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 and select Linkage from the Options menu.

7 Check Export for ActionScript if it isn't already.

8 Drag a button component from the Components panel to the Stage and 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" that 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 will have to write another handler.

Customizing the Window component

You can transform a Window component horizontally and vertically both 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

The style declaration of the title bar of a Window component is indicated by the `Window.titleStyleDeclaration` property.

A Window component supports the following Halo styles:

Style	Description
`borderStyle`	The component border; either "none", "inset", "outset", or "solid". This style does not inherit its value.

Using skins with the Window component

The Window component uses the RectBorder class which uses the ActionScript drawing API to draw its borders. You can use the `setStyle()` method (see `UIObject.setStyle()`) to change the following RectBorder style properties:

RectBorder styles	Letter
`borderColor`	a
`highlightColor`	b
`borderColor`	c
`shadowColor`	d
`borderCapColor`	e
`shadowCapColor`	f
`shadowCapColor`	g
`borderCapColor`	h

The style properties set the following positions on the border:

If you use `UIObject.createClassObject()` or `PopUpManager.createPopUp()` to create a Window instance dynamically (at runtime), you can also skin it dynamically. To skin a component at runtime, set the skin properties of the *`initObject`* parameter that is passed to the `createClassObject()` method. These skin properties set the names of the symbols to use as the button's states, both with and without an icon. For more information, see `UIObject.createClassObject()`, and `PopUpManager.createPopUp()`.

A Window component uses the following skin properties:

Property	Description
`skinTitleBackgroud`	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`.
`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`.

Window class

Inheritance UIObject > UIComponent > View > ScrollView > Window

ActionScript Class Name mx.containers.Window

The properties of the Window class allow you to set the title caption, add a close button, and set the display content at runtime. 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 non-modal. For example, the following code creates a modal Window instance (the last parameter indicates modality):

```
var newWindow = PopUpManager.createPopUp(this, Window, true);
```

Modality is simulated by creating a large transparent window underneath the Window component. Due to the way transparent windows are rendered, you may notice a slight dimming of the objects under the transparent window. The effective transparency can be set 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 closeButton on 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 only available on the class itself. The `version` property returns a string that indicates the version of the component. To access the `version` property, use the following code:

```
trace(mx.containers.Window.version);
```

Note: The following code returns undefined: `trace(myWindowInstance.version);`.

Method summary for the Window class

Method	Description
`Window.deletePopUp()`	Removes a window instance created by `PopUpManager.createPopUp()`.

Inherits all methods from UIObject, UIComponent, and View.

Property summary for the Window class

Property	Description
`Window.closeButton`	Indicates whether a close button is included on the title bar (`true`) or not (`false`).
`Window.content`	A reference to the content specified in the `contentPath` property.
`Window.contentPath`	A path to the content that is displayed in the window.
`Window.title`	The text that displays in the title bar.
`Window.titleStyleDeclaration`	The style declaration that formats the text in the title bar.

Inherits all properties from UIObject, UIComponent, and ScrollView.

Event summary for the Window class

Event	Description
`Window.click`	Broadcast when the close button is released.
`Window.mouseDownOutside`	Broadcast when the mouse is pressed outside the modal window.

Inherits all events from UIObject, UIComponent, View, and ScrollView.

Window.click

Availability

Flash Player 6 version 79.

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 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 Window component 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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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

`UIEventDispatcher.addEventListener()`, `Window.closeButton`

Window.closeButton

Availability

Flash Player 6 version 79.

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`.

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

`Window.click`, `PopUpManager.createPopUp()`

Window.content

Availability

Flash Player 6 version 79.

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

Set the value of the text property within the content inside the window component:

```
loginForm.content.password.text = "secret";
```

Window.contentPath

Availability

Flash Player 6 version 79.

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 JPG file to load. The default value is "" (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 version 79.

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 only be called 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 with the close button is clicked:

```
var myTW = PopUpManager.createPopUp(_root, Window, true);
twListener = new Object();
twListener.click = function(){
  myTW.deletePopUp();
}
myTW.addEventListener("click", twListener);
```

Window.mouseDownOutside

Availability

Flash Player 6 version 79.

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 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 Window component instance `myWindowComponent`, sends "_level0.myWindowComponent" 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, `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 a set of properties that contains information about the event. You can use these properties to write code that handles the event. Finally, you call the `UIEventDispatcher.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 about event objects, see "Event Objects" on page 562.

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

`UIEventDispatcher.addEventListener()`

Window.title

Availability

Flash Player 6 version 79.

Edition

Flash MX 2004.

Usage

windowInstance.title

Description

Property; a string indicating the caption of the title bar. The default value is `""` (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 version 79.

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

The following code creates a window that displays the content of the movie clip with the linkage identifier “ChangePassword” and uses the CSSStyleDeclaration “MyTWStyles”:

```
var myTW = PopUpManager.createPopUp(_root, Window, true,
  {contentPath:"LoginForm",
                                    titleStyleDeclaration:"MyTWStyles"});
```

For more information about styles, see “Using styles to customize component color and text” on page 27.

XMLConnector component (Flash Professional only)

The XMLConnector component is a Flash MX 2004 v2 component whose purpose is to read or write XML documents using HTTP `get` operations or `post` operations. It acts as a connector between other components and external XML data sources. The XMLConnector communicates with components in your application using either data binding features in the Flash MX Professional 2004 authoring environment, or ActionScript code. The XMLConnector component has properties, methods, and events but it has no runtime visual appearance.

The XMLConnector component and the WebServiceConnector component implement the RPC (Remote Procedure Call) Component API, a set of methods, properties, and events that define an easy way to send parameters to, and receive results from, an external data source.

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.

The schema appears in the Schema tab in the Component Inspector panel. The schema identifies the fields in the XML document that you can bind to user interface component properties in your application. You can manually create the schema through the Component Inspector panel or use the authoring environment to create one automatically.

Note: The authoring environment will accept a copy of the external XML document you are connecting to as a model for the schema. If you are familiar with XML scripting, you can create a sample XML file that can be used to generate a schema.

Although the XMLConnector component has properties and events (like other components), it has no runtime visual appearance. For more information on working with the XMLConnector component, see "The XMLConnector component (Flash Professional only)" in Using Flash Help.

XMLConnector component parameters

The following are authoring parameters that you can set for each XMLConnector component instance, in the Component Inspector panel Parameters tab:

`direction` (Enumeration) indicates whether data is being sent, received, or both.

`ignoreWhite` (Boolean type) when set to true, ignores white space when the XML is retrieved.

`multipleSimultaneousAllowed` (Boolean type) indicates whether multiple calls can take place at the same time; the default value is false.

`suppressInvalidCalls` (Boolean type) indicates whether to suppress a call if parameters are invalid; the default value is false.

`URL` (String type) is the URL of the external XML document, used in HTTP operations.

XMLConnector class (Flash Professional only)

Inheritance RPC > XMLConnector

ActionScript Class Name mx.data.components.XMLConnector

Property summary for the XMLConnector class

Property	Description
XMLConnector.direction	Indicates whether data is being sent, received, or both.
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.

Method summary for the XMLConnector class

Method	Description
XMLConnector.trigger()	Initiates a remote procedure call.

Event summary for the XMLConnector class

Event	Description
XMLConnector.result	Broadcast when a Remote Procedure Call completes successfully.
XMLConnector.send	Broadcast when the `trigger()` function is in process, after the parameter data has been gathered but before the data is validated and the Remote 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 version 79.

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 via HTTP POST 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 property 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 via HTTP GET, and valid XML data is expected from the URL.
- `send/receive` 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.multipleSimultaneousAllowed

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.multipleSimultaneousAllowed;
```

Description

Property; indicates whether multiple calls can take place at the same time. If false, then the `trigger()` function will perform a call if another call is already in progress. A `status` event will be emitted, with the code `CallAlreadyInProgress`. If true, then the call will take place.

When multiple calls are simultaneously in progress, there is no guarantee that they will complete in the same order as they were triggered. Also, Flash Player may place limits on the number of simultaneous network operations. This limit varies by version and platform.

XMLConnector.params

Availability

Flash Player 6 version 79.

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` params for `myXMLConnector`:

```
myXMLConnector.params = new XML("<mydoc><name>Bob</name><city>Oakland</city></
  mydoc>");
```

XMLConnector.result

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("result", myListenerObject);
```

Description

Event; broadcasts when an Remote Procedure Call operation successfully completes.

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 version 79.

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 storage will become available for garbage collection when the parent goes away.
- 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("send", myListenerObject);
```

Description

Event; broadcasts during the processing of a `trigger()` operation, 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 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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.addEventListener("status", myListenerObject);
```

Description

Event; broadcasts 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 will be 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 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 non-zero value NN. See the Flash Help information for XML.status to see the possible errors NN.

FaultCode	FaultString	Notes
XMLConnector.No.Data.Received	no data was received from the server	RESTRICTION: due to various browser limitations, this message can either mean (a) the server URL was invalid, not responding, or returned an HTTP error code; or (b) the server request succeeded but the response happened to be 0 bytes of data. The recommended workaround is: design your application so that the server will NEVER return 0 bytes of data. If you get "XMLConnector.No.Data.Received" , you will know for sure that there was a server error, and can inform the end-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. See Flash Help information on XML.status to see the possible errors NN.
XMLConnector.Params.Missing	Direction is 'send' or 'send/receive', but params are null.	þ

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 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.suppressInvalidCalls;
```

Description

Property; indicates whether to suppress a call if parameters are invalid. If `true`, then the `trigger()` function will not perform a call if the bound parameters fail the validation. A "status" event will be emitted, with the code `InvalidParams`. If `false`, then the call will take place, using the invalid data as required.

XMLConnector.trigger()

Availability

Flash Player 6 version 79.

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 will 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).

XMLConnector.URL

Availability

Flash Player 6 version 79.

Edition

Flash MX Professional 2004.

Usage

```
componentInstance.URL;
```

Description

Property; the URL that this component uses when doing HTTP operations. This URL may be either an absolute or relative URL. The URL is subject to all the standard Flash Player security protections.

XUpdateResolver component (Flash Professional only)

You use resolver components in combination with the DataSet component (part of the data management functionality in the Macromedia Flash data architecture). The resolver components enable you to convert changes made to the data within your application into a format that is appropriate for the external data source that you are updating. These components have no visual appearance at runtime.

If you use a DataSet component in your application, it generates an optimized set of instructions (DeltaPacket) that describes the changes made to the data at runtime. This set of instructions that is converted to the appropriate format (update packet) by the resolver components. When an update is sent to the server, the server sends a response (result packet) containing additional updates or errors that result from the update operation. The resolver components can convert this information back into a DeltaPacket that can be applied to the DataSet component to keep it in sync with the external data source. Resolver components enable you to keep your application and an external data source in sync without writing additional ActionScript code.

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 or XHive. The XUpdateResolver component translates the DeltaPacket into XUpdate statements. An external data source can process these XUpdates statements. The XML document contains the necessary information and formatting for updating any standard XUpdate database.

For information about the working draft of the XUpdate language specification, see www.xmldb.org/xupdate/xupdate-wd.html. A parallel resolver component, RDBMSResolver (see "RDBMSResolver component (Flash Professional only)" on page 436), exists for returning data to an XML-based server. For more information about DataSet components, see "DataSet component (Flash Professional only)" on page 193. For more information about connectors, see "WebServiceConnector (Flash Professional only)" on page 604 and "XMLConnector component (Flash Professional only)" on page 624. For more information about the Flash data architecture, see "Resolver components (Flash Professional only)" in Using Flash Help.

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.

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 XUpdate packet consists of an optimized set of instructions that describe the inserts, edits, and deletes performed on the DataSet component. The resolver component gets a DeltaPacket from a DataSet component, sends its own XUpdate packet to a connector, receives server errors back from the connection, and communicates them back to the DataSet component—all using bindable properties.

Using the XUpdateResolver component (Flash Professional only)

Use this XUpdateResolver 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 XML-formatted data source.

For more information on working with the XUpdateResolver component, see "Resolver components (Flash Professional only)" in Using Flash Help.

XUpdateResolver component parameters

includeDeltaPacketInfo *Boolean*; if `true`, causes the XUpdate to include additional information from the `deltaPacket` in attributes on the XUpdate nodes. This information includes the transaction ID and operation ID.

The following example shows how the update packet is constructed when the Boolean value for this property 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 example shows how the update packet is constructed when the Boolean value for this property 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>
```

Property summary for the XUpdateResolver component

Property	Description
`XUpdateResolver.deltaPacket`	Contains a description of the changes to the DataSet component.
`XUpdateResolver.includeDeltaPacketInfo`	Includes additional information from the `deltaPacket` in attributes on the XUpdate nodes.
`XUpdateResolver.updateResults`	Describes results of update.
`XUpdateResolver.xupdatePacket`	Contains the XUpdate translation of the changes to the DataSet component.

Event summary for the XUpdateResolver component

Event	Description
`XUpdateResolver.beforeApplyUpdates`	Called by the resolver component to make custom modifications immediately after the XML packet has been created and immediately prior to that packet being 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 resolver event object should contain the following properties:

Property	Description
target	Object; resolver firing this event.
type	String; name of the event.
updatePacket	XML object; XML object 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 deltaPacket, and immediately prior to that packet being 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; property of type `deltaPacket` that receives a `deltaPacket` to be translated into an `xupdatePacket`, and outputs a `deltaPacket` from any server results placed into 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 `deltaPacket` again so it can be resent the next time the `deltaPacket` 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 modified before being added to the next `deltaPacket`.

XUpdateResolver.includeDeltaPacketInfo

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

`resolveData``.includeDeltaPacketInfo`

Description

Property; property of type `Boolean` that, if `true`, includes additional information from the `deltaPacket` in attributes on the XUpdate nodes. This information will includes the transaction ID and operation ID.

For an example of the resulting XML, see "XUpdateResolver component parameters" on page 633.

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 `updatePacket`s. This resolver event object should contain the following properties:

Property	Description
`target`	Object; resolver firing this event.
`type`	String; name of the event.

Returns

None.

Description

Event; use this callback to insert any code after the results have been received from the server and immediately prior to the transmission, through data binding, of the `deltaPacket` containing operation results. This is a good place to put code that handles messages from the server.

Example

The following example reconciles two `updatePacket`s 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 `deltaPacket` again so it can be resent the next time the `deltaPacket` 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 modified before being added to the next `deltaPacket`.

XUpdateResolver.xupdatePacket

Availability

Flash Player 7.

Edition

Flash MX Professional 2004.

Usage

resolveData.xupdatePacket

Description

Property; property of type `xml` contains the XUpdate translation of the changes to the DataSet component. Bind this to the connector component's property that transmits the translated update packet of changes back to the DataSet component.

CHAPTER 5
Creating Components

This chapter describes how to create your own components, make them usable by other developers, and package them for deployment.

What's new

The current version (version 2) of the Macromedia Component Architecture is very different from the Macromedia Flash MX version (version 1). Macromedia made changes to improve scalability, performance, and extensibility of components for developers. The following list provides an overview of some of the changes:

- Component Inspector panel that recognizes ActionScript metadata
- Managers and base classes you can extend
- Built-in Live Preview
- Improved compiler messages
- New event model
- Focus management
- CSS-based styles

Working in the Flash environment

The Macromedia Flash MX 2004 and Flash MX Professional 2004 environment is set up to make the structure of classes and components logical. This section describes where you should store your component files.

FLA file assets

When creating a component, you start with a FLA file and add skins, graphics, and other assets. You can store these assets anywhere in the FLA file, because Flash component users need only a compiled component file and not the source assets.

You use two-frame, two-layer SWF files when creating components in Flash. The first layer is an actions layer, which points to the component's ActionScript class file. The second layer is an assets layer, which contains graphics, symbols, and other assets used by the component.

Class files

The FLA file includes a reference to the ActionScript class file for the component. This is known as binding the component to the class file.

The ActionScript code specifies the properties and methods for the component, and defines which, if any, classes your component inherits from. You must use the *.as file naming convention for ActionScript source code and name the source code file after the component itself. For example, MyComponent.as contains the source code for the MyComponent component.

The Flash MX 2004 core class .as files reside within a single folder called Classes/mx/Core. Other ActionScript class files are organized by package names in their own folders under /Classes.

For a custom component, create a new directory under /Classes and store the ActionScript class file there.

The classpath

This section describes the Flash classpath.

Understanding the classpath

The classpath is an ordered list of directories in which Flash searches for class files during component export or SWF file generation. The order of the classpath entries is important because Flash uses the classes on a first-come, first-served basis. At export time, classes found on the classpath that match linkage identifiers in the FLA file are imported into the FLA file and registered with their symbols.

A global classpath refers to all FLA files generated with Flash. A local classpath applies only to the current FLA file.

The default local classpath is empty. The default global classpath consists of the following two paths:

- $(UserConfig)/Classes (Macintosh); $(LocalData)/Classes (Windows)
- .

The dot (.) indicates the current working directory. Flash searches the FLA file's current directory for ActionScript classes.

The $(UserConfig)/Classes and $(LocalData)/Classes paths indicate the per-user configuration directory. This directory points to the following locations:

- In Windows, this directory is c:\Documents and Settings*username*\Application Data\Macromedia\Flash MX 2004\en\Configuration.
- On the Macintosh, this directory is *volume*:Users:*username*:Library:Application Support:Macromedia:Flash MX 2004:en:configuration.

The UserConfig and LocalData directories mirror the directories located in *Flash_root*/en/Configuration. However, the classpath does not directly include those directories, and it is relative to the UserConfig or LocalData directory.

Changing the classpath

You can change the classpath for an individual FLA file (local classpath) or for all FLA files you work with in Flash (global classpath).

To change the global classpath:

1 Select Edit > Preferences.

The Preferences dialog box appears.

2 Select the ActionScript tab.

3 Click the ActionScript 2.0 Settings button.

The ActionScript Settings dialog box appears.

4 Add, remove, or edit entries in the Classpath box.

5 Save your changes.

To change the local classpath:

1 Select File > Publish Settings.

The Publish Settings dialog box appears.

2 Select the Flash tab.

3 Click the Settings button.

The ActionScript Settings dialog box appears.

4 Add, remove, or edit entries in the Classpath box.

5 Save your changes.

Locating component source files

When developing a component, you can store the source files in any directory. However, you must include that directory in the Flash MX 2004 classpath settings to ensure that Flash finds the necessary class files when exporting the component. In addition, to test the component, you must store the component in the Flash Components directory. For more information on storing SWC files, see "Using SWC files" on page 661.

Editing symbols

Each symbol has its own Timeline. You can add frames, keyframes, and layers to a symbol Timeline, just as you can to the main Timeline.

When creating components, you start with a symbol. Flash provides the following three ways for you to edit symbols:

- Edit the symbol in the context of the other objects on the Stage by using the Edit in Place command. Other objects are dimmed to distinguish them from the symbol you are editing. The name of the symbol you are editing is displayed in an edit bar at the top of the Stage, to the right of the current scene name.
- Edit a symbol in a separate window by using the Edit in New Window command. Editing a symbol in a separate window lets you see the symbol and the main Timeline at the same time. The name of the symbol you are editing is displayed in the edit bar at the top of the Stage.
- Edit the symbol by changing the window from the Stage view to a view of only the symbol, using symbol-editing mode. The name of the symbol you are editing is displayed in the edit bar at the top of the Stage, to the right of the current scene name.

Examples of component code

Flash MX 2004 and Flash MX Professional 2004 include the following component source files to help you develop your own components:

- FLA file source code: *Flash MX 2004_install_dir*/en/First Run/ComponentFLA/StandardComponents.fla
- ActionScript class files: *Flash MX 2004_install_dir*/en/First Run/Classes/mx

Creating components

This section describes how to create a component that subclasses an existing Flash MX 2004 class. Subsequent sections describe how to write the component's ActionScript class file and edit the component for usability and quality.

Creating a new component symbol

All components are MovieClip objects, which are a type of symbol. 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 radio button for the behavior.

A MovieClip object has its own multiframe Timeline that plays independently of the main Timeline.

5 Click the Advanced button.

The advanced settings appear in the dialog box.

6 Select Export for ActionScript. This tells Flash to package the component by default with any Flash content that uses the component.

7 Enter a linkage identifier.

This identifier is used as symbol name, linkage name, and as the associated class name.

8 In the AS 2.0 Class text box, enter the fully qualified path to the ActionScript 2.0 class, relative to your classpath settings.

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).

For more information on setting the Flash MX 2004 classpath, see "Understanding the classpath" on page 640.

9 In most cases, you should deselect Export in First Frame (it is selected by default). For more information, see "Best practices when designing a component" on page 663.

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.

You can now edit this symbol and add it to your component's FLA file.

Editing symbol layers

Once you have created the new symbol and defined 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.

For information on how to edit symbols, see "Editing symbols" on page 641.

To edit symbol layers:

1 Enter symbol-editing mode.

2 Rename an empty layer, or create a layer called Actions.

3 In the Actions panel, add a single line that imports the component's fully qualified ActionScript class file.

This statement relies on the Flash MX 2004 classpath settings. (For more information, see "Understanding the classpath" on page 640.) The following example imports the MyComponent.as file that is in the package myPackage:

```
import myPackage.MyComponent;
```

Note: Use the `import` statement and not the `include` statement when importing an ActionScript class file. Do not surround the class name or package with quotation marks.

4 Rename an empty layer, or create a layer called Assets.

The Assets layer includes all the assets used by this component.

5 In the first frame, add a `stop()` action in the Actions panel, as the following example shows:

```
stop();
```

Do not add any graphical assets to this frame. Flash Player will stop before the second frame, in which you can add the assets.

6 If you are extending an existing component, locate that component and any other base classes that you use, and place an instance of that symbol in your layer's second frame. To do this, select the symbol from the Components panel and drag it onto the Stage in the second frame of your component's Assets layer.

Any asset a component uses (whether it's another component or media such as bitmaps) should have an instance placed inside the component.

7 Add any graphical assets used by this component on the second frame of your component's Assets layer. For example, if you are creating a custom button, add the graphics that represent the button's states (up, down, and so on).

8 When you have finished creating the symbol content, do one of the following to return to document-editing mode:

- Click the Back button at the left side of the edit bar above the Stage.
- Select Edit > Edit Document.
- Click the scene name in the edit bar above the Stage.

Adding parameters

The next step in component development is to define the component parameters. Parameters are the primary method by which users modify instances of the components you create.

In previous editions of Flash, you defined the parameters using the Component Inspector panel. In Flash MX 2004 and Flash MX Professional 2004, you define parameters in the ActionScript class file, and the Component Inspector panel discovers which ones are public and displays them to users.

The next section deals with writing the component's external ActionScript file, which includes information on adding component parameters.

Writing the component's ActionScript

Most components include some kind of ActionScript code. The type of component determines where you will write your ActionScript and how much ActionScript to write. There are two basic approaches to component development:

- Creating new components with no parent classes
- Extending existing component classes

This section focuses on extending existing components. If you are creating a component that derives from another component's class file, you should write an external ActionScript class file as described in this section.

Extending existing component classes

When creating a component symbol that derives from a parent class, you link it to an external ActionScript 2.0 class file. (For information on defining this file, see "Creating a new component symbol" on page 642.)

The external ActionScript class extends another class, adds methods, adds getters and setters, and defines event handlers for the component. To edit ActionScript class files, you can use Flash, any text editor, or any Integrated Development Environment (IDE).

You can inherit from only one class. ActionScript 2.0 does not allow multiple inheritance.

Simple example of a class file

The following is a simple example of a class file called MyComponent.as. This example contains a minimal set of imports, methods, and declarations for a component that inherits from the UIObject class.

```
import mx.core.UIObject;

  class myPackage.MyComponent extends UIObject {
  static var symbolName:String = "MyComponent";
  static var symbolOwner:Object = Object(myPackage.MyComponent);
  var className:String = "MyComponent";
  #include "../core/ComponentVersion.as"
  function MyComponent() {
  }
  function init(Void):Void {
    super.init();
  }
  function size(Void):Void {
```

```
        super.size();
    }
}
```

General process for writing a class file

Use the following general process when writing the ActionScript for a component. Some steps may be optional, depending on the type of component you create.

This process is covered in more detail in the rest of this chapter.

To write the ActionScript file for a component:

1 Import all necessary classes.

2 Define the class using the `class` keyword; extend a parent class, if necessary.

3 Define the `symbolName` and `symbolOwner` variables; these are the symbol name of your ActionScript class and the fully qualified package name of the class, respectively.

4 Define your class name as the `className` variable.

5 Add versioning information.

6 Enter your default member variables.

7 Create variables for every skin element/linkage used in the component. This lets users set a different skin element by changing a parameter in the component.

8 Add class constants.

9 Add a metadata keyword and declaration for every variable that has a getter/setter.

10 Define uninitialized member variables.

11 Define getters and setters.

12 Write a constructor. It should generally be empty.

13 Add an initialization method. This method is called when the class is created.

14 Add a size method.

15 Add custom methods or override inherited methods.

Importing classes

The first line of your external ActionScript class file should import necessary class files that your class uses. This includes classes that provide functionality, as well as the superclass your class extends, if any.

You import the fully qualified class name, rather than the filename of the class, when using the `import` statement, as the following example shows:

```
import mx.core.UIObject;
import mx.core.ScrollView;
import mx.core.ext.UIObjectExtensions;
```

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.

Selecting a parent class

Most components have some common behavior and functionality. Flash includes two base classes to supply this commonality. By subclassing these classes, your components begin with a basic set of methods, properties, and events.

The following table briefly describes the two base classes:

Full 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. It can participate in tabbing, accept low-level events such as keyboard and mouse input, and be disabled so it does not receive mouse and keyboard input. UIComponent provides the following functionality: • Creating focus navigation • Enabling and disable components • Resizing components

Understanding the UIObject class

Components based on version 2 of the Macromedia Component Architecture descend from the UIObject class, which wraps the MovieClip class. The MovieClip class is the base class for all classes in Flash that can draw on the screen. Many MovieClip properties and methods are related to the Timeline, which is an unfamiliar tool to developers who are new to Flash. The UIObject class was created to abstract many of those details. Subclasses of MovieClip do not document unnecessary MovieClip properties and methods. However, you can access these properties and methods if you want.

UIObject tries to hide the mouse handling and frame handling in MovieClip. It posts events to its listeners just before drawing (the equivalent of `onEnterFrame`), when loading and unloading, and when its layout changes (`move`, `resize`).

UIObject provides alternate read-only variables for determining the position and size of the movie clip. You can use the `move()` and `setSize()` methods to alter the position and size of an object.

Understanding the UIComponent class

The UIComponent class is a child of UIObject. It is the base class of all components that have user interaction (mouse and keyboard input).

Extending other classes

To make component construction easier, you can subclass any class; you are not required to extend the UIObject or UIComponent class directly. If you extend any other component's class, you extend these classes by default. Any component class listed in the Component dictionary can be extended to create a new component class.

Flash includes a group of classes that draw on the screen and inherit from UIObject. For example, the Border class draws borders around other objects. Another example is RectBorder, which is a subclass of Border and knows how to resize its visual elements appropriately. All components that support borders should use one of the border classes or one of the border subclasses. For a detailed description of these classes, see Chapter 4, "Components Dictionary," on page 43.

For example, if you want to create a component that behaves almost exactly the same as a Button component does, you can extend the Button class instead of recreating all the functionality of the Button class from the base classes.

Writing the constructor

Constructors are methods that have a unique purpose: to set properties and perform other tasks when a new instance of a component is instantiated. You can recognize a constructor because it has the same name as the component class itself. For example, the following code shows the ScrollBar subcomponent's constructor:

```
function ScrollBar() {
}
```

In this case, when a new scroll bar is instantiated, the `ScrollBar()` constructor is called.

Generally, component constructors should be empty so that the object can be customized with its properties interface. In addition, setting properties in constructors can sometimes lead to overwriting default values, depending on the ordering of initialization calls.

A class can contain only one constructor function; overloaded constructor functions are not allowed in ActionScript 2.0.

Versioning

When releasing components, you should define a version number. This lets developers know whether they should upgrade, and helps with technical support issues. When setting a component's version number, use the static variable `version`, as the following example shows:

```
static var version:String = "1.0.0.42";
```

If you create many components as part of a component package, you can include the version number in an external file. Thus, you update the version number in only one place. For example, the following code imports the contents of an external file that stores the version number in one place:

```
#include "../myPackage/ComponentVersion.as"
```

The contents of the ComponentVersion.as file are identical to the above variable declaration, as the following example shows:

```
static var version:String = "1.0.0.42";
```

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` properties in your component's ActionScript class file.

The following table describes these variables:

Variable	Description
`symbolName`	Symbol name for the object. This variable is static and of type String.
`symbolOwner`	Class used in the internal call to the `createClassObject()` method. This value should be the fully qualified class name, which includes the package's path. This variable is static and of type Object.
`className`	Name of the component class. This variable is also used in calculating style values. If `_global.styles[className]` exists, it sets defaults for a component. This variable is of type String.

The following example shows a custom component's naming:

```
static var symbolName:String = "MyComponent";
static var symbolOwner:Object = custom.MyComponent;
var className:String = "MyComponent";
```

Defining getters and setters

Getters and setters provide visibility to component properties and control access to those properties by other objects.

The convention for defining getter and setter methods is to precede the method name with `get` or `set`, followed by a space and the property name. It's a good idea to use initial capital letters for each word that follows the `get` or `set`.

The variable that stores the property's value cannot have the same name as the getter or setter. By convention, precede the name of the getter and setter variables with two underscores.

The following example shows the declaration of `initialColor`, and getter and setter methods that get and set the value of this property:

```
...
public var __initialColor:Color = 42;
...
public function get initialColor():Number {
  return __initialColor;
}
public function set initialColor(newColor:Number) {
  __initialColor = newColor;
}
```

Getters and setters are commonly used in conjunction with metadata keywords to define properties that are visible, are bindable, and have other properties.

Component metadata

Flash recognizes component metadata statements in your external ActionScript class files. The metadata tags can define component attributes, data binding properties, and events. Flash interprets these statements and updates the development environment accordingly. This allows you to define these members once, rather than in the ActionScript code and the development panels.

The metadata tags can only be used in external ActionScript class files. You cannot use metadata tags in the action frames of your FLA files.

Using metadata keywords

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. 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 metadata keywords 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 panel, 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:

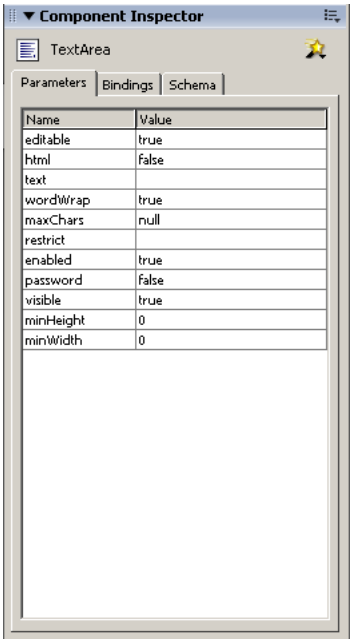
Tag	Description
Inspectable	Defines an attribute exposed to component users in the Component Inspector panel. See “Inspectable” on page 649.
InspectableList	Identifies which subset of inspectable parameters should be listed in the Property inspector. If you don't add an InspectableList attribute to your component's class, all inspectable parameters appear in the Property inspector. See “InspectableList” on page 651.
Event	Defines component events. See “Event” on page 652.
Bindable	Reveals a property in the Bindings tab of the Component Inspector panel. See “Bindable” on page 652.
ChangeEvent	Identifies events that cause data binding to occur. See “ChangeEvent” on page 653.
IconFile	The filename for the icon that represents this component in the Flash Components panel. See “Adding an icon” on page 662.

The following sections describe the component metadata tags in more detail.

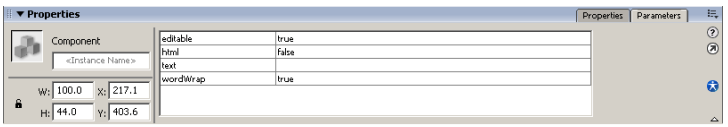
Inspectable

You specify the user-editable (or “inspectable”) parameters of a component in the class definition for the component, and these parameters appear in the Component Inspector panel. 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 in the Component Inspector panel.

The following figure shows the Parameters tab in the Component Inspector panel for the Text Area control:



Alternatively, you can view a subset of the component properties on the Property inspector Parameters tab, as the following figure shows:



When determining which parameters to reveal in the authoring environment, Flash uses the Inspectable metadata keyword. The syntax for this keyword 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 keyword also supports loosely typed attributes like this:

```
[Inspectable("danger", 1, true, maybe)]
```

The metadata statement must immediately precede the property's variable declaration to be bound to that property.

The following table describes the attributes of the Inspectable metadata keyword:

Attribute	Type	Description
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 • Object • List • String • Number • Boolean • Font Name • Color
defaultValue	String or Number	(Required) A default value for the inspectable property.
enumeration	String	(Optional) Specifies a comma-delimited list of legal values for the property.
verbose	Number	(Optional) Indicates that this inspectable property should be displayed only when the user indicates that verbose properties should be included. If this attribute is not specified, Flash assumes that the property should be displayed.
category	String	(Optional) Groups the property into a specific subcategory in the Property inspector.
listOffset	Number	(Optional) Added for backward compatibility with Flash MX components. Used as the default index into a List value.
variable	String	(Optional) Added for backward compatibility with Flash MX components. Used to specify the variable that this parameter is bound to.

InspectableList

Use the InspectableList metadata keyword to specify exactly which subset of inspectable parameters should appear in the Property inspector. Use InspectableList in combination with Inspectable so that you can hide inherited attributes for subclassed components. If you do not add an InspectableList metadata keyword 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 keyword 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 DotParent class:

```
[InspectableList("flavorStr","colorStr")]
class BlackDot extends DotParent {
  [Inspectable(defaultValue="strawberry")]
  public var flavorStr:String;
  [Inspectable(defaultValue="blue")]
  public var colorStr:String;
  ...
}
```

Event

Use the Event metadata keyword to define events that this component emits.

The syntax for this keyword is as follows:

```
[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 they 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 {
  ...
}
```

Bindable

Data binding connects components to each other. You achieve visual data binding through the Bindings tab of the Component Inspector panel. 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. The Component Inspector panel is the main tool used in Flash MX Professional 2004 to do data binding.

Use the Bindable metadata keyword to make properties and getter/setter functions in your ActionScript classes appear in the Bindings tab in the Component Inspector panel.

The Bindable metadata keyword has the following syntax:

```
[Bindable[readonly|writeonly[,type="datatype"]]]
```

The Bindable keyword must precede a property, getter/setter function, or other metadata keyword that precedes a property or getter/setter function.

The following example defines the variable `flavorStr` as a public, inspectable variable that is also accessible on the Bindings tab of the Component Inspector panel:

```
[Bindable]
[Inspectable(defaultValue="strawberry")]
public var flavorStr:String = "strawberry";
```

The Bindable metadata keyword 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`	Instructs Flash to allow the property to be only the source of a binding, as shown in this example: `[Bindable("readonly")]`
`writeonly`	Instructs Flash to allow the property to be only the destination of a binding, as shown in this example: `[Bindable("writeonly")]`
`type="`*`datatype`*`"`	Specifies the data type of the property that is being bound. If you do not specify this option, data binding uses the property's data type as declared in the ActionScript code. If *`datatype`* is a registered data type, you can use the functionality in the Schema tab's Data Type pop-up menu. The following example sets the data type of the property to String: `[Bindable(type="String")]`

You can combine the access option and the data type option, as the following example shows:

```
[Bindable(param1="writeonly",type="DataProvider")]
```

The Bindable keyword is required when you use the ChangeEvent metadata keyword. For more information, see "ChangeEvent" on page 653.

For information on creating data binding in the Flash authoring environment, see "Data binding (Flash Professional only)" in Using Flash Help.

ChangeEvent

Use the ChangeEvent metadata keyword to generate one or more component events when changes are made to bindable properties.

The syntax for this keyword is as follows:

```
[Bindable]
[ChangeEvent("event"[,...)]
property_declaration or get/set function
```

Like Bindable, this keyword can be used only with variable declarations or getter and setter functions.

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 specified in the metadata occurs, Flash informs whatever is bound to the property that the property has changed.

You can also instruct your component to generate an event when a getter or setter function is called, as the following example shows:

```
[Bindable]
[ChangeEvent("change")]
function get selectedDate():Date
  ...
```

In most cases, you set the `change` event on the getter, and dispatch the event on the setter.

You can register multiple `change` events in the metadata, as the following example shows:

```
[ChangeEvent("change1", "change2", "change3")]
```

Any one of those events indicates a change to the variable. They do not all have to occur to indicate a change.

Defining component parameters

When building a component, you can add parameters that define its appearance and behavior. The most commonly used properties appear as authoring parameters in the Component Inspector panel. You define these properties by using the Inspectable keyword (see "Inspectable" on page 649). You can also set all inspectable parameters with ActionScript. Setting a parameter with ActionScript overrides any value set during authoring.

The following example sets several component parameters in the JellyBean class file, and exposes them with the Inspectable metadata keyword in the Component Inspector panel:

```
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;
}
```

Parameters can be any of the following supported types:

- Array
- Object
- List
- String
- Number
- Boolean

- Font Name
- Color

Implementing core methods

There are two core methods that must be implemented by all components: the size and initialization methods. If you do not override these two methods in a custom component, Flash Player might produce an error.

Implementing the initialization method

Flash calls the initialization method when the class is created. At a minimum, the initialization method should call the superclass's initialization method. The `width`, `height`, and `clip` parameters are not properly set until after this method is called.

The following sample initialization method from the Button class calls the superclass's initialization method, sets the scale and other default property values, and gets the value for the color attribute from the UIObject object:

```
function init(Void):Void {
  super.init();
  labelField.selectable = false;
  labelField.styleName = this;
  useHandCursor = false;
  // mark as using color "color"
  _color = UIObject.textColorList;
}
```

Implementing the size method

Flash calls the component's size method from the `setSize()` method to lay out the contents of the component. At a minimum, the size method should call the superclass's size method, as the following example shows:

```
function size(Void):Void {
  super.size();
}
```

Handling events

Events allow a component to know when the user has interacted with the interface, and also to know when important changes have occurred in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing.

The event model is a dispatcher/listener model based on the XML Events specification. You write code that registers listeners with the target object so that when the target object dispatches an event the listeners are called.

Listeners are either functions or objects, but not methods. The listener receives a single event object as its parameter that contains the name of the event and includes all relevant information about the event.

Components generate and dispatch events and consume (listen to) other events. An object that wants to know about another object's events registers with that object. When an event occurs, the object dispatches the event to all registered listeners by calling a function requested during registration. To receive multiple events from the same object, you must register for each event.

Flash MX 2004 extends the ActionScript `on()` handler to support component events. Any component that declares events in its class file and implements the `addEventListener()` method is supported.

Common events

Following is a list of common events broadcast by various classes. Every component should try to 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	Parameters	Use
`click`	None	Used by Button, or whenever a mouse click has no other meaning.
`scroll`	`Scrollbar.lineUp`, `lineDown`, `pageUp`, `pageDown`, `thumbTrack`, `thumbPosition`, `endScroll`, `toTop`, `toBottom`, `lineLeft`, `lineRight`, `pageLeft`, `pageRight`, `toLeft`, `toRight`	Used by ScrollBar and by other controls that cause scrolling (scroll "bumpers" on a scrolling pop-up menu).
`change`	None	Used by List, ComboBox, and other text entry components.
`maxChars`	None	Used when user tries to enter too many characters in text entry components.

In addition, because of inheritance from UIComponent, 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 emit focus, but all emit 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 has been 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 has been released.

Using the event object

An event object is passed to a listener as a parameter. The event object is an ActionScript object whose properties contain information about the event that occurred. You can use the event object in 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 trace the value:

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

The following table lists the properties that are common to every event object:

Property	Description
`type`	A string that indicates the name of the event. This property is required.
`target`	A reference to the component instance that is broadcasting the event. In general, you are not required to describe this reference object explicitly.

The most common events, such as `click` and `change`, have no required properties other than `type`.

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 dispatches the event in a single line:

```
ancestorSlide.dispatchEvent({type:"revealChild", target:this});
```

In the preceding example, setting the `target` property is optional, since 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 4, "Components Dictionary," on page 43.

Dispatching events

In the body of your component's ActionScript class file, you broadcast events using the `dispatchEvent()` method. The signature for the `dispatchEvent()` method is as follows:

```
dispatchEvent(eventObj)
```

The `eventObj` parameter is the event object that describes the event (see "Using the event object" on page 657.)

Identifying event handlers

You define the event handler object or event handler function that listens for your component's events in your application's ActionScript.

The following example creates a listener object, handles a `click` event, and adds it as an event listener to myButton:

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

In addition to using a listener object, you can use a function as a listener. A listener is a function if it does not belong to an object. For example, the following code creates the listener function `myHandler()` and registers it to myButton:

```
function myHandler(eventObj){
  if (eventObj.type == "click"){
    // your code here
  }
}
myButton.addEventListener("click", myHandler);
```

For more information on using the `addEventListener()` method, see "Using component event listeners" on page 22.

When you know that a particular object is the only listener for an event, you can take advantage of the fact that the new event model always calls a method on the component instance. This method is the event name plus the word `Handler`. For example, to handle the `click` event, write the following code:

```
myComponentInstance.clickHandler = function(o){
  // insert your code here
}
```

In the above code, the keyword `this`, if used in the callback function, is scoped to myComponentInstance.

You can also use listener objects that support a `handleEvent()` method. Regardless of the name of the event, the listener object's `handleEvent()` method is called. You must use an `if...else` or a `switch` statement to handle multiple events, which makes this syntax clumsy. For example, the following code uses an `if...else` statement to handle the `click` and `enter` events:

```
myObj.handleEvent = function(o){
  if (o.type == "click"){
    // your code here
  } else if (o.type == "enter"){
    // your code here
  }
}
target.addEventListener("click", myObj);
target2.addEventListener("enter", myObj);
```

Using the Event metadata

Add Event metadata in your ActionScript class file for each event listener. The value of the Event keyword becomes the first argument in calls to the `addEventListener()` method, as the following example shows:

```
[Event("click")] // event declaration
...
class FCheckBox{
  function addEventListener(eventName:String, eventHandler:Object) {
    ... //eventName is String
  }
}
```

For more information on the Event metadata keyword, see "Event" on page 652.

Skinning

A user interface (UI) control is composed entirely of attached movie clips. This means that all assets for a UI control can be external to the UI control movie clip, so they can be used by other components. For example, if your component needs button functionality, you can reuse the existing Button component assets.

The Button component uses a separate movie clip to represent each of its states (FalseDown, FalseUp, Disabled, Selected, and so on). However, you can associate your 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. This ability to change skins during authoring as well as runtime is called *skinning.*

To use skinning in components, create a variable for every skin element/linkage used in the component. This lets someone set a different skin element just by changing a parameter in the component, as the following example shows:

```
var falseUpSkin = "mySkin";
```

The name "mySkin" is subsequently used as the linkage name of the MovieClip symbol to display the false up skin.

The following example shows the skin variables for the various states of the Button component:

```
var falseUpSkin:String  = "ButtonSkin";
var falseDownSkin:String  = "ButtonSkin";
var falseOverSkin:String = "ButtonSkin"
var falseDisabledSkin:String = "ButtonSkin";
var trueUpSkin:String = "ButtonSkin";
var trueDownSkin:String = "ButtonSkin";
var trueOverSkin:String = "ButtonSkin";
var trueDisabledSkin:String = "ButtonSkin";
var falseUpIcon:String = "";
var falseDownIcon:String = "";
var falseOverIcon:String = "";
var falseDisabledIcon:String = "";
var trueUpIcon:String = "";
var trueDownIcon:String = "";
var trueOverIcon:String = "";
var trueDisabledIcon:String = "";
```

Adding styles

Adding styles is the process of registering all the graphic elements in your component with a class and letting that class control the color schemes of graphics at runtime. No special code is necessary in the component implementations to support styles. Styles are implemented entirely in the base classes and skins.

For more information about styles, see "Using styles to customize component color and text" on page 27.

Making components accessible

A growing requirement for web content is that it should be accessible to people who have disabilities. Visually impaired people can use visual content in Flash applications by means of screen reader software, which provides an audio description of the material on the screen.

Flash MX 2004 includes the following accessibility features:

- Custom focus navigation
- Custom keyboard shortcuts
- Screen-based documents and the screen authoring environment
- The Accessibility class

When you create a component, you can include ActionScript that enables the component and a screen reader to communicate. Then, when developers use your component to build an application in Flash, they use the Accessibility panel to configure each component instance.

Add the following line to your component's FLA file, in the same layer that you add other ActionScript calls:

```
mx.accessibility.ComponentName.enableAccessibility();
```

For example, the following line enables accessibility for the MyButton component:

```
mx.accessibility.MyButton.enableAccessibility();
```

When developers add the MyButton component to an application, they can use the Accessibility panel to make it accessible to screen readers.

For information on the Accessibility panel and other accessibility features of Flash, see "Creating Accessible Content" in Using Flash Help.

Exporting the component

Flash MX 2004 exports components as component packages (SWC files). When you distribute a component, you only need to give your users the SWC file. This file contains all the code, SWF files, images, and metadata associated with the component so that users can easily drop it into their Flash environment.

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.
Source code	If the component is created with Flash MX 2004, the source code is one or more ActionScript files that contain a class declaration for the component. The source code is used only for type checking when subclassing components and is not compiled by the authoring tool since 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.
Implementing 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 implementing SWF files 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).
Debug info	(Optional) A SWD file corresponding to the implementing SWF file. The filename is always the same as that of the SWF file, but with the extension .swd. If it is included in the SWC file, debugging of the component is allowed.
Icon	(Optional) A PNG file containing the 18 x 18, 8-bit-per-pixel icon used to display a component in the authoring tool user interface(s). If no icon is supplied, a default icon is displayed. (See "Adding an icon" on page 662.)
Property inspector	(Optional) If specified, this SWF file is used as a custom Property inspector in the authoring tool. If omitted, the default Property inspector is displayed to the user.

To view the contents of a SWC file, you can open it using any compression utility that supports PKZip format (including WinZip).

You can optionally include other files in the SWC file, once you have generated 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.

Multiple SWC files are expanded into a single directory, so each component must have a unique filename to prevent conflicts.

Using SWC files

This section describes how to create and import SWC files. You should give instructions for importing SWC files to your component users.

Creating SWC files

Flash MX 2004 and Flash MX Professional 2004 provide the ability to create SWC files by exporting a movie clip as a SWC file. When creating a SWC file, Flash reports compile-time errors as if you were testing a Flash application.

To export a SWC file:

1 Select an item in the Flash library.
2 Right-click (Windows) or Control-click (Macintosh) the item and select Export SWC File.
3 Save the SWC file.

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 use a SWC file in the Flash authoring environment:

1 Close the Flash authoring environment.
2 Copy the SWC file into the *flash_root*/en/First Run/Components directory.
3 Start the Flash authoring environment or reload the Components panel.
The component's icon should appear in the Components panel.

Making the component easier to use

Once you have created the component and prepared it for packaging, you can make it easier for your users to use. This section describes some techniques for adding usability to your component.

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 be saved 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.

Using 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, at their approximate size.

Adding a Live Preview is no longer necessary when creating components using the v2 architecture. Component SWC files include the implementing SWF file, and the component uses that SWF file on the Flash Stage.

Adding tooltips

Tooltips appear when a user rolls the mouse over your component name or icon in the Components panel of the Flash authoring environment.

To add tooltips to your component, use the `tiptext` keyword outside the class definition in the component's ActionScript class file. You must comment out this keyword using an asterisk (*) and precede it with an @ symbol for the compiler to recognize it properly.

The following example shows the tooltip for the CheckBox component:

```
* @tiptext Basic CheckBox component. Extends Button.
```

Best practices when designing a component

Use the following practices when designing a component:

- Keep the file size as small as possible.
- Make your component as reusable as possible by generalizing functionality.
- Use the new event model rather than the `on(event)` syntax.
- Use the Border class rather than graphical elements to draw borders around objects.
- Use tag-based skinning.
- Use the `symbolName` property.
- 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.

INDEX

D

E

F

G

N

O

P

R

S

T

U

V

W

X