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: August 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
- PopUpManager
- 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
- Alert component
- DataBinding package
- DateField component
- DataGrid component
- DataHolder component
- DataSet component
- DateChooser component
- Form class
- MediaController component
- MediaDisplay component
- MediaPlayback component
- Menu component
- RDBMSResolver component
- Screen class

- Tree component
- WebServiceConnector component
- XMLConnector component
- XUpdateResolver component

**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:

- HD/Applications/Macromedia Flash MX 2004/First Run/Components (Macintosh)
- \Program Files\Macromedia\Flash MX 2004\<language>\First Run\Components (Windows)

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 *Using Flash*, the *ActionScript Reference Guide*, 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 the 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 the *ActionScript Reference Guide* and ActionScript Dictionary Help.

# 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 v1 components to v2 architecture" on page 25.

## Categories of components

Components included with Flash MX 2004 and Flash MX Professional 2004 fall into four categories: user interface (UI) controls, containers, data, and managers. UI controls allow a user to interact with an application; for example, the RadioButton, CheckBox, and TextInput components are UI controls. Containers are shells for different types of content, such as loaded SWF files and JPEG files; the ScrollPane and Window components are containers. Data components allow you to load and manipulate information from data sources; the WebServiceConnector and XMLConnector components are data components. Managers are non-visual 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. For a complete list of each category, see Chapter 4, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

## 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 v1 components to v2 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. 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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.**

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 21.)

**Skin properties** allow you to load states only when needed. (See "About skinning components" on page 37.)

**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 28.)

**Themes** allow you to drag a new look onto a set of components. (See "About themes" on page 35.)

**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 240 and "UIObject" on page 250.)

**Packaging as a SWC file** allows easy distribution and concealable code. See "Creating Components". You may need to download the latest PDF from the Flash Support Center to see this information.

**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 allows you to create unique namespaces, import classes as needed, and subclass easily to extend components. See "Creating Components" and the *ActionScript Reference Guide*. You may need to download the latest PDF from the Flash Support Center to see this information.

## 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 "Creating Components". You may need to download the latest PDF from the Flash Support Center to see this information.

## 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, check its entry in Chapter 4, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45. 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. You may need to update your Help system to see this information.

Most components built by Macromedia are also navigable by the keyboard. Each component's entry in Chapter 4, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45 indicates whether or not you can control the component with the keyboard.

# CHAPTER 2
# Working with Components

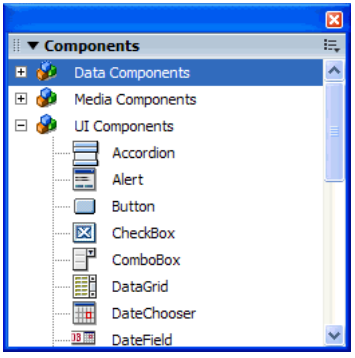
There are various ways to work with components in Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004. You use the Components panel to view components and add them to a document during authoring. Once a component has been added to a document, you can view its properties in the Property 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/First Run/Components (Macintosh) or Macromedia\Flash 2004\<language>\First Run\Components (Windows) folder are displayed in the Components panel.

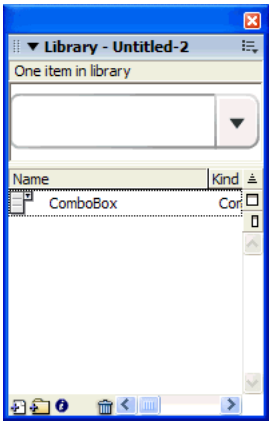
**To display the Components panel:**

- 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 symbol (SWC) 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

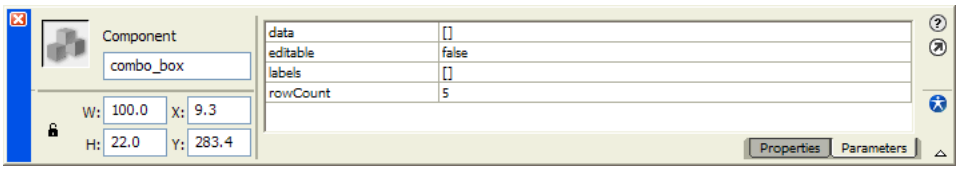
After you add an instance of a component to a Flash document, you use the Property inspector to set and view information for the instance. You create an instance of a component by dragging it from the Components panel onto the Stage; then you name the instance in the Property inspector and specify the 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.

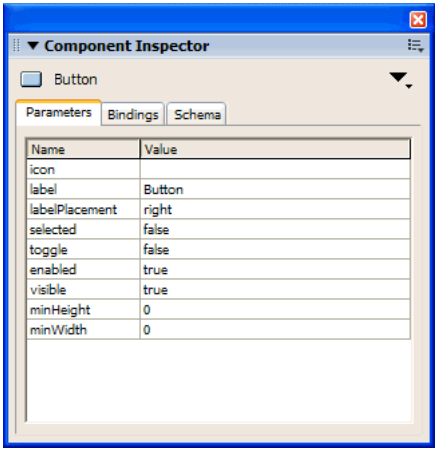


**To view parameters for a component instance in the Component Inspector panel:**

1 Select Window > Component Inspector.

2 Select an instance of a component on the Stage.

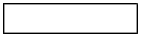
3 To view parameters, click the Parameters tab.




## 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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45. 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 "Creating Components". You may need to download the latest PDF from the Flash Support Center to see this information.

## 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 "Creating Components". You may need to download the latest PDF from the Flash Support Center to see this information.

**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 Component Parameters 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 Component Parameters panel.

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

1 Select Window > 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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

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 28.

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

- Apply a theme (see "About themes" on page 35).
- Edit a component's skins (see "About skinning components" on page 37).

## 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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

***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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

## 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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

## 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:

```
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 component event handlers

The easiest way to handle a component event is to use the `on()` component event handler. You can assign the `on()` handler to a component instance, just as you would assign a handler to a button or movie clip.

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() 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("CheckBox was clicked");
}
```

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

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

For more information, see each event entry in Chapter 4, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

## 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 248.

For information about the events a component broadcasts, see each component's entry in Chapter 4, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

**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 only be used 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 "Creating Components". You may need to download the latest PDF from the Flash Support Center to see this information.

## Creating custom focus navigation

When a user presses the Tab key to navigate in a Flash application or clicks in an application, the FocusManager 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 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. 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;
```

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 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. The DepthManager application programming interface (API) allows you to place user interface (UI) 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" on page 96).

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 v1 components to v2 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. 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 "Creating Components". You may need to download the latest PDF from the Flash Support Center to see this information.

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" in ActionScript Dictionary 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 33.

***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 30.

**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 31.

**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 29.

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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

**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" on page 214.

***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" on page 214). 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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

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 29, "Setting styles for specific components" on page 30, and "Setting styles for a component class" on page 31.

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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

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). |
| 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). |

| Style | Description |
|---|---|
| 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:

- Windows—First Run\ComponentFLA
- Macintosh—First Run/ComponentFLA

You can create new themes and apply them to an application to change the look and feel of all the components. For example, you could create 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. The ScrollBar, for instance, is programmed to load the symbol with the linkage identifier `ScrollDownArrowDown` as the skin for the pressed (down) state of its down arrow. 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 28.

### 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 edit symbol 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 edit document 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 35. 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 component is made up of three skins: ScrollDownArrowDisabled, ScrollDownArrowUp, and ScrollDownArrowDown. Some components share skins. Components that use scroll bars—including ComboBox, List, ScrollBar, 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 35).

  ***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 "Editing component skins" on page 38, 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 42).

***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 edit symbol 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 edit document mode.
9. Select File > Save but don't close MyTheme.fla. Now you must create a new document in which to apply the edited skin to a component.

   For more information, see "Applying 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 42. For information about how to apply a new skin, see "About skinning components" on page 37.

***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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

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

1 Follow the steps in "Editing component skins" on page 38, 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 38, 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 edit symbol 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 38, 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, "Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components," on page 45.

**To skin a subcomponent, do the following:**

1 Follow the steps in "Editing component skins" on page 38, 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 checkbox 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
# Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 Components

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) controls

| Component | Description |
|---|---|
| Accordion component | A set of vertical overlapping views with buttons along the top that allow users to switch views. |
| Alert component | 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 editable text field at the top of the list that allows users to search the list. |
| DateChooser component | Allows users to choose a date or dates from a calendar. |
| DateField component | A uneditable text field with a calendar icon. When a user clicks anywhere inside the bounding box of the component, a DateChooser component is displayed. |
| DataGrid component | 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. |
| 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. |
| Menu component | Allows users to select one command from a list; a standard desktop application menu. |
| 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 choose between mutually exclusive options. |
| ScrollBar component | Allows users to control the portion of data that is displayed when there is too much information to fit in the display area. |
| TextArea component | An optionally editable, multiline text field. |
| TextInput component | An optionally editable, single-line text input field. |
| Tree component | Allows a user to manipulate hierarchical information. |

## Containers

| Component | Description |
|---|---|
| Accordion component | Displays content in vertical overlapping panes with buttons along the top that allow you to switch views. |
| Loader component | A container that holds a loaded SWF or JPEG file. |
| ScrollPane component | Displays movies, bitmaps, and SWF files in a limited area using automatic scroll bars. |
| Window component | A draggable window with a title bar, caption, border, and close button that display content to the user. |

## Data

| Component | Description |
|---|---|
| DataBinding package | Theses classes implement the Flash runtime data binding functionality. |
| DataHolder component | Holds data and can be used as a connector between components. |
| DataProvider component | This component is the model for linear-access lists of data. This model provides simple array-manipulation capabilities that broadcast their changes. |
| DataSet component | A building block for creating data-driven applications. |
| RDBMSResolver component | 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. |
| WebServiceConnector component | Provides scriptless access to web service method calls. |
| XMLConnector component | Reads and writes XML documents using the HTTP GET and POST methods. |
| XUpdateResolver component | Allows you to save data back to any supported data source. This resolver component translates the DeltaPacket into XUpdate. |

## Managers

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

## Screens

| Component | Description |
|---|---|
| Slide class | Allows you to manipulate slide presentation screens at runtime. |
| Form class | Allows you to manipulate form application screens at runtime. |

## Accordion component

For the latest information about this feature, click the Update button at the top of the Help tab.

## Alert component

For the latest information about this feature, click the Update button at the top of the Help tab.

## 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" on page 102.

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. You may need to update your Help system to see this information.

## 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. For more information, see "Using the Button component" on page 49.

**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 28.

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. |
| `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 37, 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. |
| 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 Namespace** 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.

### 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 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 249.

#### 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.

#### 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.

### 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

`Window.titleStyleDeclaration`, `SimpleButton.emphasized`

## Button.icon

### Availability

Flash Player 6.

### 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 edit symbols 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.

### 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.

### 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.

### 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.

#### 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 interface

For the latest information about this feature, click the Update button at the top of the Help tab.

## 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" on page 102.

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. You may need to update your Help system to see this information.

### 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 229.

## 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 28.

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. |
| `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 37.

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 Namespace** 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);`.

### 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.

### Method summary for the CheckBox class

Inherits all methods 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.

### 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 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 249.

### 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.

### 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.

### Edition

Flash MX 2004.

### Usage

*checkBoxInstance*.labelReplacement

### 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 62.

### 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.

#### 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 229), 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. |
| 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. |
| 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. |
| 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. |
| 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" on page 102.

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. You may need to update your Help system to see this information.

## 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 choose 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 28.

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 50.)
- The text is a TextInput instance and uses its styles. (See "Using styles with the TextInput component" on page 232.)
- The drop-down list is an List instance and uses its styles. (See "Using styles with the List component" on page 117.)

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. |

## 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 37.

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 Namespace** 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.

#### 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.

### 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.

### 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 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 249.

### 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.

### 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.

### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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 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 249.

### 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.

### 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.

#### 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 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 249.

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.

### 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 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 249.

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.

### 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.

### 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.

### 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.

### 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.

### 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 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 249.

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.

### 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.

#### 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.

#### 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.

### 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.

### 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 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 249.

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.

### 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.

### 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.

### 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.

### 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.

#### 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);
```

## DataBinding package

For the latest information about this feature, click the Update button at the top of the Help tab.

## DataGrid component

For the latest information about this feature, click the Update button at the top of the Help tab.

## DataHolder component

For the latest information about this feature, click the Update button at the top of the Help tab.

## DataProvider component

For the latest information about this feature, click the Update button at the top of the Help tab.

## DataSet component

For the latest information about this feature, click the Update button at the top of the Help tab.

## DateChooser component

For the latest information about this feature, click the Update button at the top of the Help tab.

## DateField component

For the latest information about this feature, click the Update button at the top of the Help tab.

## DepthManager

**ActionScript class namespace** 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.

### 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.

### 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.

### 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.

### 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.

#### 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.

### 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.

### 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 Dictionary Help.

## FocusManager

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 Focus Manager 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

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;
```

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(){
  trace("button 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 namespace** 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.

**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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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

For the latest information about this feature, click the Update button at the top of the Help tab.

## 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. If the text is too long, and you choose to set the autoSize parameter, the autoSize parameter is not supported by the live preview and the label's bounding box is not resized. You must click within the boundary box to select the label on the Stage.

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. You may need to update your Help system to see this information.

### 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, do the following:
   - 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 110. 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 28.

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 37.

## Label class

**Inheritance** UIObject > Label

**ActionScript Class Namespace** 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.

#### 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.

### 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.

### 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" on page 102.

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. You may need to update your Help system to see this information.

## 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 choose 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** 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 199.

## 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. |
| 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. |

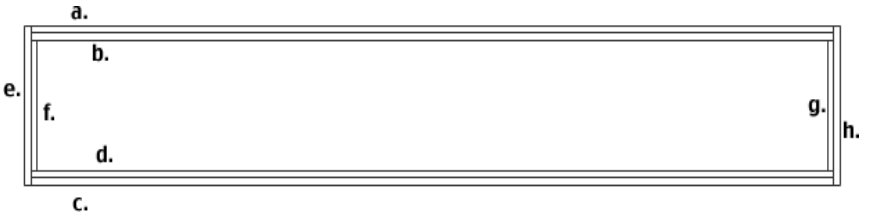
A List component also uses the style properties of the Label component (see "Using styles with the Label component" on page 111), the ScrollBar component (see "ScrollBar component" on page 188), and RectBorder.

### Using skins with the List component

All the skins in the List component are included in the subcomponents from which the list is composed (ScrollBar component and RectBorder). For more information, see "ScrollBar component" on page 188. 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 Namespace** 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, highlighted, 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. |
| `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 cellRenderer to use for 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.

#### 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.

### 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.

### 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 for the cell renderer to use. Any class used for this property must implement the "CellRenderer interface" on page 59.

### Example

The following example uses a linkage identifier to set a new cell renderer:

```
myList.cellRenderer = "ComboBoxCell";
```

## List.change

### Availability

Flash Player 6.

### 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 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 249.

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.

#### 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 component" on page 96.

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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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 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 249.

### 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.

### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

#### 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.

### 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.

### 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.

### 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.

### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

#### 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.

#### 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" on page 102.

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. You may need to update your Help system to see this information.

## 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.

**content** 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 the stand-alone 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 do the following:
   - 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

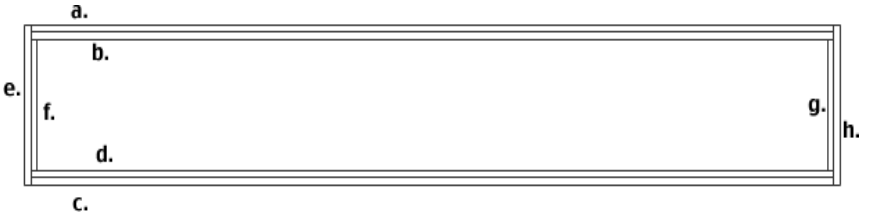
The Loader component doesn't use styles.

### 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 |
|---|
| `borderColor` |
| `highlightColor` |
| `borderColor` |
| `shadowColor` |
| `borderCapColor` |
| `shadowCapColor` |
| `shadowCapColor` |
| `borderCapColor` |

The style properties set the following positions on the border:




## Loader class

**Inheritance** UIObject > UIComponent > View > Loader

**ActionScript Class Namespace** 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.

### 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.

### 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.

#### 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.

#### 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 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 249.

### 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.

### 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.

### 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 the stand-alone 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.

### 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.

#### 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.

### 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 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 249.

#### 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.

#### 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;
```

## MediaController component

For the latest information about this feature, click the Update button at the top of the Help tab.

## MediaDisplay component

For the latest information about this feature, click the Update button at the top of the Help tab.

## MediaPlayback component

For the latest information about this feature, click the Update button at the top of the Help tab.

## Menu component

For the latest information about this feature, click the Update button at the top of the Help tab.

## 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" on page 102.

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. You may need to update your Help system to see this information.

## 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 28.

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 37.

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. |
| 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 properties 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 properties from UIObject and UIComponent.

## NumericStepper.change

#### Availability

Flash Player 6.

#### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

#### 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

**ActionScript class namespace** 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.

### 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.

### 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 142.

**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.content = "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 28.

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 37.

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 Namespace** 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.

#### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

#### 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 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 249.

### 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.

### 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.

### 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.

#### 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" on page 102.

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 179.

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. You may need to update your Help system to see this information.

## 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 262).

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 28.

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 37.

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 37. 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. |
| `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.

### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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

For the latest information about this feature, click the Update button at the top of the Help tab.

## RemoteProcedureCall interface

For the latest information about this feature, click the Update button at the top of the Help tab.

## Screen class

For the latest information about this feature, click the Update button at the top of the Help tab.

## ScrollBar component

The ScrollBar component is a standard user interface element that provides scrolling functionality to other components. The scroll bar consists of four parts: two arrow buttons, a track, and a thumb (the button that slides up and down the track). The position of the thumb and the display of the buttons depends on the current state of the scroll bar. The scroll bar uses four parameters to calculate its display state: a minimum range value (`ScrollBar.minPos`), a maximum range value (`ScrollBar.maxPos`), a current position that must be within the `minPos` and `maxPoss` values, and a viewport size (ScrollBar. which must be equal to or less than the range and represents the number of items in the range that can be displayed at once.

The user uses the mouse to click the various portions of the scrollbar which dispatches events to listeners. The object listening to the component is responsible for updating the portion of data displayed. The ScrollBar will update itself to represent the new state after the action has taken place.

There are five places a user can press a scroll bar, and each broadcasts its own notification. The following table describes the results of user interaction with the scroll bar:

| Position | Event |
|---|---|
| Up arrow/Left arrow | Broadcasts a `scroll` event with the `detail` property set to `"LineUp"` (vertical placement) or `"LineLeft"` (horizontal placement). |
| Right arrow/Down arrow | Broadcasts a `scroll` event with the `detail` property set to `"LineDown"` (vertical placement) or `"LineRight"` (horizontal placement). |
| Track above or to the left of the thumb. | Broadcasts a `scroll` event with the `detail` property set to `"PageUp"` (vertical placement) or `"PageLeft"` (horizontal placement). |
| Track above or to the left of the thumb. | Broadcasts a `scroll` event with the `detail` property set to `"PageDown"` (vertical placement) or `"PageRight"` (horizontal placement). |
| Thumb | Broadcasts a `scroll` event with the `detail` property set to `"ThumbTrack"` when the mouse is pressed on the thumb, and broadcasts a `scroll` event with the `detail` property set to `"ThumbPosition"` when the mouse is released. If the thumb is positioned at the very top or left of the track, another `scroll` event is generated with the `detail` property set to `"AtTop"` (vertical placement) or `"AtLeft"` (horizontal placement). If the thumb is positioned at the very bottom or right of the track, another `scroll` event is generated with the `detail` property set to `"AtBottom"` (vertical placement) or `"AtRight"` (horizontal placement). |

The ScrollBar does not display correctly if it is smaller than the height of the up arrow and down arrow buttons; you can prevent this by hiding the scroll bar. If there is not enough room for the thumb, the thumb is hidden.

## Using the ScrollBar component

The ScrollBar component is used primarily for adding scrolling capability to other components. You can use this component if you are creating a new component that requires a scroll bar. There are two scroll bar components, VScrollBar (vertical) and HScrollBar (horizontal).

### ScrollBar parameters

The ScrollBar doesn't have its own authoring parameters. It is always a part of another component.

### Creating an application with the ScrollBar component

The following procedure explains how to add a ScrollBar component to a component while authoring. The component will display a row of images and needs a scroll bar to allow users to move images right and left along the row.

**To add a ScrollBar component to another component, do the following:**

1 Open the new component symbol in edit symbol mode.

2 Drag an HScrollBar component from the Components panel to the Stage.

3 In the Property inspector, enter the instance name **hSB**.

4 Select Frame 1 in the Timeline, open the Actions panel, and enter the following code that creates a listener object for the scroll bar that moves the images along the row:

```
scrollListener = new Object();
scrollListener.scroll = function (evt){
  // image_mc is an instance of the Loader component
  image_mc.contentPath = arrayOfPicture[hSB.scrollPosition];
}
hSB.addEventListener("scroll", scrollListener);
```

## Customizing the ScrollBar component

You can transform a ScrollBar 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()`.

When you resize a scroll bar, the track and the thumb change size but the arrow buttons don't. Also, the track and thumb hit areas are resized but the button hit areas do not change. If there isn't enough room for the thumb, it is hidden. If the scroll bar is too small, it may not display correctly. To check the minimum size, use the `minHeight` property for VScrollBar and the `minWidth` property for HScrollBar.

### Using styles with the ScrollBar component

You can set style properties to change the appearance of a scroll 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 28.

A ScrollBar component supports the following Halo styles:

| Style | Description |
|---|---|
| `themeColor` | The background color for the scroll bar. Possible values are "haloGreen", "haloBlue", and "haloOrange". |

### Using skins with the ScrollBar component

The ScrollBar component uses the movie clip symbols to display its states. To skin the ScrollBar 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/ScrollBar Elements folder in the library of HaloTheme.fla or SampleTheme.fla. For more information, see "About skinning components" on page 37.

If you use the `UIObject.createClassObject()` method to create a ScrollBar 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 `UIObject.createClassObject()` method. These skin properties set the names of the symbols to use as the states of the buttons, track, and thumb of the scroll bar.

A ScrollBar component uses the following skin properties:

| Property | Description |
|---|---|
| scrollTrackName | The expandable middle of the track. The default value is ScrollTrack. |
| scrollTrackOverName | The track when the pointer is over it (this skin is optional). The default value is ScrollTrack. |
| scrollTrackDownName | The track when the mouse is pressed (this skin is optional). The default value is ScrollTrack. |
| upArrowName | The up arrow button that contains its disabled state. The default value is BtnUpArrow. |
| upArrowUpName | The up arrow button in its neutral state. The default value is ScrollUpArrowUp. |
| upArrowDownName | The up arrow button when pressed. The default value is ScrollUpArrowDown. |
| upArrowOverName | The up arrow button when the pointer is over it. The default value is ScrollUpArrowOver. |
| downArrowName | The down arrow button that contains the disabled state. |
| downArrowUpName | The down arrow button in its neutral state. |
| downArrowDownName | The down arrow button in its pressed state. |
| downArrowOverName | The down arrow button in its over state. (optional) |
| thumbTopName | The fixed-size top edge of the thumb. |
| thumbBottomName | The fixed-size bottom edge of the thumb. |
| thumbTopDownName | The fixed-size top edge of the thumb when pressed (optional). |
| thumbBottomDownName | The fixed-size bottom edge of the thumb when pressed (optional). |
| thumbTopOverName | The fixed-size top edge of the thumb when the pointer is over it (optional). |
| thumbBottomOverName | The fixed-size bottom edge of the thumb when the pointer is over it (optional). |
| thumbMiddleName | The expandable middle of the thumb. |
| thumbMiddleDownName | The expandable middle of the thumb when pressed (optional). |
| thumbMiddleOverName | The expandable middle of the thumb when the pointer is over it (optional). |
| thumbGripName | The grip portion of the thumb. |
| thumbGribDownName | The grip portion of the thumb when pressed (optional). |
| thumbGripOverName | The grip portion of the thumb when the pointer is over it (optional). |

## ScrollBar class

**Inheritance** UIObject > UIComponent > ScrollBar > VScrollBar; UIObject > UIComponent > ScrollBar > HScrollBar

**ActionScript Class Namespace** mx.controls.VScrollBar; mx.controls.HScrollBar

The properties of the ScrollBar class allow you to set a scrolling range and indicate the amount that the view content should scroll at runtime.

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

### Method summary for the ScrollBar class

| Method | Description |
|---|---|
| `ScrollBar.setScrollProperties()` | Sets the scroll range and the display area size. |

Inherits all methods from UIObject and UIComponent.

### Property summary for the ScrollBar class

| Property | Description |
|---|---|
| `ScrollBar.lineScrollSize` | The amount to scroll when the arrow button is pressed. |
| `ScrollBar.minPos` | The lower range for the scroll bar. |
| `ScrollBar.minHeight` | The minimum height for the scroll bar. This property is read-only. |
| `ScrollBar.minWidth` | The minimum width of the scroll bar. This property is read-only. |
| `ScrollBar.maxPos` | The upper range of the scroll bar. |
| `ScrollBar.pageScrollSize` | The amount to scroll when the track is pressed. |
| `ScrollBar.pageSize` | The size of the display area. |
| `ScrollBar.scrollPosition` | The current scroll position. |

Inherits all methods from UIObject and UIComponent.

### Event summary for the ScrollBar class

| Event | Description |
|---|---|
| `ScrollBar.scroll` | Broadcast when any part of the scroll bar is pressed. |

Inherits all methods from UIObject and UIComponent.

## ScrollBar.lineScrollSize

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*scrollBarInstance*.lineScrollSize

### Description

Property; a number indicating the increment to move the content when an arrow button is pressed. The default value is 1.

### Example

The following code moves the content in increments of 5:

```
hSB.lineScrollSize = 5;
```

## ScrollBar.maxPos

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*scrollBarInstance*.maxPos

### Description

Property; sets the upper range of the scroll bar. This property can be set in the *initObject* parameter of the UIObject.createClassObject() method. The default value is 0.

### Example

The following code passes the value of maxPos as a parameter of the createClassObject() method:

```
createClassObject(HScrollBar, "hSB", 0, {minPos:0, maxPos:100, pageSize:10});
```

### See also

ScrollBar.minPos, UIObject.createClassObject()

## ScrollBar.minPos

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*scrollBarInstance*.minPos

### Description

Property; sets the lower range for the scroll bar. This property can be set in the *initObject* parameter of the UIObject.createClassObject() method. The default value is 0.

### Example

The following code passes the value of minPos as a parameter of the createClassObject() method:

```
createClassObject(HScrollBar, "hSB", 0, {minPos:0, maxPos:100, pageSize:10});
```

### See also

ScrollBar.maxPos, UIObject.createClassObject()

## ScrollBar.minHeight

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*scrollBarInstance*.minHeight

### Description

Property (read-only); returns the minimum height of the scroll bar in pixels. The size is determined by the skins used to display the scroll bar.

### Example

The following code explains how to size a horizontal scroll bar:

```
hSB.setSize(100, hSB.minHeight);
```

### See also

UIObject.setSize()

## ScrollBar.minWidth

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*scrollBarInstance*.minWidth

### Description

Property (read-only); returns the minimum width of the scroll bar in pixels. The size is determined by the skins used to display the scroll bar.

### Example

The following code explains how to size a horizontal scroll bar:

```
hSB.setSize(minWidth, 100);
```

### See also

UIObject.setSize()

## ScrollBar.pageScrollSize

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*scrollBarInstance*.pageScrollSize

### Description

Property; the increment to move the display area when the track is pressed. This value is reset to the *pageSize* parameter of ScrollBar.setScrollProperties() whenever that method is called. The default value is 0.

### Example

This example moves the display area by sets of 50 units when the scroll track is pressed. The second line of code checks to see if the property is defined, as in the following:

```
hSB.pageScrollSize = 50;
if (hSB.pageScrollSize != undefined){
  trace("defined");
}
```

## ScrollBar.pageSize

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*scrollBarInstance*.pageSize

### Description

Property; the page size (the size of the display area) for the scroll bar. This property can be set in the *initObject* parameter of the UIObject.createClassObject() method. The default value is 0.

### Example

The following code passes the value of minPos as a parameter of the createClassObject() method:

```
createClassObject(HScrollBar, "hSB", 0, {minPos:0, maxPos:100, pageSize:10});
```

### See also

UIObject.createClassObject()

## ScrollBar.scroll

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

Usage 1:

```
on(scroll){
  ...
}
```

Usage 2:

```
listenerObject = new Object();
listenerObject.scroll = function(eventObject){
  ...
}
scrollBarInstance.addEventListener("scroll", listenerObject)
```

### Event Object

In addition to the standard event object properties, there is a detail property defined for the scroll event. For a list of possible values, see "ScrollBar component" on page 188.

### 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 ScrollBar 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 `sb`, sends "_level0.sb" to the Output panel:

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

The second usage example uses a dispatcher/listener event model. A component instance (*`scrollBarInstance`*) dispatches an event (in this case, `scroll`) and the event is handled by 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 249.

### Example

This example creates a `form` listener object with a `scroll` callback function that sends a message to the Output panel with the value of the scroll position:

```
form.scroll = function(eventObj){
  // eventObj.target is the component which generated the change event,
  // i.e., the ScrollBar.
  trace("now viewing picture #:" + eventObj.target.scrollPosition);
}
hSB.addEventListener("scroll", form);
```

### See also

`UIEventDispatcher.addEventListener()`

## ScrollBar.scrollPosition

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*`scrollBarInstance`*`.scrollPosition`

#### Description

Property; the current scroll position. Setting this property updates the position of the thumb. The default value is 0.

#### Example

The following code moves to the first item in the list and then tests to see if the location is the first thing in the list:

```
hSB.scrollPosition = 0;
if (hSB.scrollPosition == 0){
  trace("position is 0");
}
```

### ScrollBar.setScrollProperties()

#### Availability

Flash Player 6.

#### Edition

Flash MX 2004.

#### Usage

*scrollBarInstance*.setScrollProperties(*pageSize*, *minPos*, *maxPos*)

#### Parameters

*pageSize* A number of items viewable in the display area.

*minPos* A number indicating the lowest numbered item in the scroll range.

*maxPos* A number indicating the highest numbered item in the scroll range.

#### Returns

A number indicating the amount of progress that had been made.

#### Description

Method; sets the scroll range and the display area size for the scroll bar. The scroll bar updates the state of the buttons and the size of the thumb according to the values of the parameters.

#### Example

This example sets the display area to show 10 items at a time out of a range of 0 to 99:

```
hSB.setScrollProperties(10, 0, 99);
```

# 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" on page 102.

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 scrollbar is turned off, the vertical scrollbar is displayed from top to bottom along the right side of the scroll pane. When the vertical scrollbar is turned off, the horizontal scrollbar 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. For more information, see "Using styles with the ScrollBar component" on page 190.

### Using skins with the ScrollPane component

The ScrollPane component doesn't have any of its own skins, but the scroll bars that it uses do have skins. For more information, see "Using skins with the ScrollBar component" on page 190.

## ScrollPane class

**Inheritance** UIObject > UIComponent > View > ScrollView > ScrollPane

**ActionScript Class Namespace** 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.

### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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 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 249.

### 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.

### 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.

#### 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 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 249.

### 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.

### 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.

### 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.

### Edition

Flash MX 2004.

*`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.

#### 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.

#### 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";
```

## StyleManager

**ActionScript class namespace** 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.

### 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.

### 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.

#### 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");
```

## Slide class

For the latest information about this feature, click the Update button at the top of the Help tab.

## 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" on page 102.

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 229. 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 the 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 28.

A TextArea component supports the following styles:

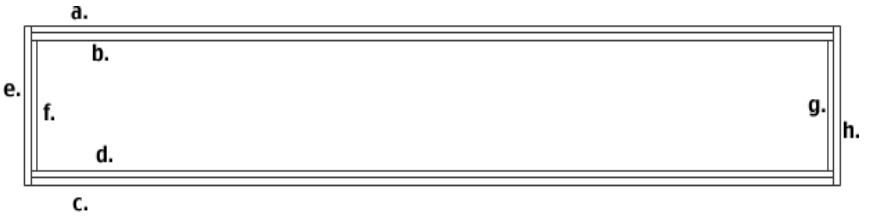
| Style | Description |
|---|---|
| color | The default color for text. |
| embedFonts | The fonts to embed in the document. |
| fontFamily | The font name for text. |
| fontSize | The point size for the font. |
| fontStyle | The font style, either "normal",or "italic". |
| fontWeight | The font weight, either "normal" or "bold". |
| 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 |
|---|
| borderColor |
| highlightColor |
| borderColor |
| shadowColor |
| borderCapColor |
| shadowCapColor |
| shadowCapColor |
| borderCapColor |

The style properties set the following positions on the border:



## TextArea class

**Inheritance** UIObject > UIComponent > View > ScrollView > TextArea

**ActionScript Class Namespace** 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.

### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

#### 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" on page 102.

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 216. 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:

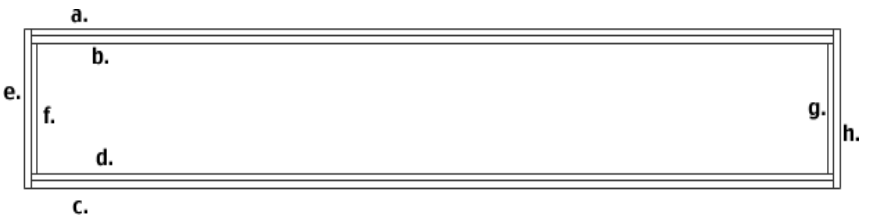
| Style | Description |
|---|---|
| `color` | The default color for text. |
| `embedFonts` | The fonts to embed in the document. |
| `fontFamily` | The font name for text. |
| `fontSize` | The point size for the font. |
| `fontStyle` | The font style, either "normal",or "italic". |
| `fontWeight` | The 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 |
|---|
| `borderColor` |
| `highlightColor` |
| `borderColor` |
| `shadowColor` |
| `borderCapColor` |
| `shadowCapColor` |
| `shadowCapColor` |
| `borderCapColor` |

The style properties set the following positions on the border:



## TextInput class

**Inheritance** UIObject > UIComponent > TextInput

**ActionScript Class Namespace** 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" on page 102.

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` | Triggered when the Input field changes. |
| `TextInput.enter` | Triggered when the enter key is pressed. |

Inherits all methods from UIObject and UIComponent.

## TextInput.change

#### Availability

Flash Player 6.

#### 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 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 249.

#### 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.

#### 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.

#### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

#### 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"
```

## Tree component

For the latest information about this feature, click the Update button at the top of the Help tab.

## UIComponent

**Inheritance** UIObject > UIComponent

**ActionScript class namespace** 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.

### 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 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 249.

### 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.

### 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 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 249.

### 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.

### 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.

### 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.

### 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 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 249.

### 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.

### 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 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 249.

### 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.

### 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.

### 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 namespace** 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 21.

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.

### 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);
```

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 249.

### 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 namespace** 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. 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 | Returns the position of the bottom edge of the object relative to the bottom edge of its parent. |
| UIObject.height | The height of the object in pixels. |
| UIObject.left | The left position of the object in pixels. |
| UIObject.right | The position of the right edge of the object relative to the right edge of its parent. |
| 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. |
| UIObject.visible | A Boolean value indicating whether the object is visible (`true`) or not (`false`). |
| UIObject.width | The width of the object in pixels. |
| UIObject.x | The left position of the object in pixels. |
| UIObject.y | Returns the position of the top edge of the object relative to its parent. |

## Event summary for the UIObject class

| Event | Description |
|---|---|
| UIObject.draw | Broadcast when an object is about to draw its graphics. |
| 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.unload | Broadcast when the subobjects are being unloaded. |

# UIObject.bottom

### Availability

Flash Player 6.

### 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.

### Example

Sets the value of tmp to the bottom position of the check box:

```
var tmp = checkbox.bottom;
```

## UIObject.createObject()

### Availability

Flash Player 6.

### 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.

### 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.

#### 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.

### 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 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 249.

### 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.

### Edition

Flash MX 2004.

### Usage

*componentInstance*.height

### Description

Property (read-only); a number indicating the height of the object in pixels.

### Example

The following example sets the value of `tmp` to the height of the `checkbox` instance:

```
var tmp = checkbox.height;
```

## UIObject.getStyle()

### Availability

Flash Player 6.

### 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.

### 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.

### Edition

Flash MX 2004.

### Usage

*componentInstance*.left

### Description

Property (read-only); a number indicating the left edge of the object in pixels.

### Example

The following example sets the value of `tmp` to the left position of the `checkbox` instance:

```
var tmp = checkbox.left; // sets value of tmp to left position of checkbox;
```

## UIObject.load

### Availability

Flash Player 6.

### 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 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 249.

### 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.

### 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 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 249.

### Example

The following example calls the `move()` method to keep `form2` 100 pixels down and to the right of `form1`:

```
formListener.handleEvent = function(eventObj)
{
  // eventObj.target is the component that generated the change event
  form2.move(form1.x + 100, form1.y + 100);
}
form1.addEventListener("move", formListener);
```

## UIObject.move()

### Availability

Flash Player 6.

### 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.

Failure to follow these rules may result in fuzzier-looking controls.

### Example

This example moves the ProgressBar instance pBar to the upper left corner at 100, 100:

```
pBar.move(100, 100);
```

## UIObject.redraw()

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*componentInstance*.redraw()

### Parameters

*always* A Boolean; true if redraw always, false if redraw only if invalidated.

### Returns

Nothing.

### Description

Method; forces validation of the object so it draws in the current frame

### Example

This example forces the `pBar` instance to redraw immediately:

```
pBar.validate(true);
```

## UIObject.resize

### Availability

Flash Player 6.

### 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 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, resize) and the event is handled by 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 249.

### 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.right

### Availability

Flash Player 6.

### 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.

### Example

The following example sets the value of `tmp` to the right position of the `checkbox` instance:

```
var tmp = checkbox.right;
```

## UIObject.scaleX

### Availability

Flash Player 6.

### 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.

### 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.

#### 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.

#### 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.

#### 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.

#### Edition

Flash MX 2004.

#### Usage

*componentInstance*.top

#### Description

Property (read-only); a number indicating the top edge of the object in pixels.

#### Example

The following example sets the `tmp` variable to the top position of the `checkbox` instance:

```
var tmp = checkbox.top; // sets value of tmp to top position of checkbox;
```

## UIObject.unload

#### Availability

Flash Player 6.

#### 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 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 249.

### 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.

### 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.

### Edition

Flash MX 2004.

### Usage

*componentInstance*.width

### Description

Property (read-only); a number indicating the width of the object in pixels.

### Example

The following example sets the width of the TextArea component to 450 pixels:

```
mytextarea.width = 450;
```

## UIObject.x

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*componentInstance*.x

### Description

Property (read-only); a number indicating the left edge of the object in pixels.

### Example

The following example sets the left edge of the check box to 150:

```
checkbox.x = 150;
```

## UIObject.y

### Availability

Flash Player 6.

### Edition

Flash MX 2004.

### Usage

*componentInstance*.y

### Description

Property (read-only); a number indicating the top edge of the object in pixels

#### Example

The following example sets the top edge of the check box to 200:

```
checkbox.y = 200;
```

## WebServices package

For the latest information about this feature, click the Update button at the top of the Help tab.

## WebServiceConnector component

For the latest information about this feature, click the Update button at the top of the Help tab.

## 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.

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 in focus ordering. For more information about controlling focus, see "Creating custom focus navigation" on page 24 or "FocusManager" on page 102.

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. You may need to update your Help system to see this information.

### 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 use 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. Use the PopUpManager to create modal windows that overlap other objects on the Stage. For more information, see Window class.

## 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 Button 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**.
2. In the library, select the PasswordForm movie clip and select Linkage from the Options menu.
3. Check Export for ActionScript and enter **PasswordForm** in the Identifier box.
4. Enter **mx.core.View** in the class field.
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.
8. Open the Actions panel, and enter the following click handler on Frame 1:

```
buttonListener = new Object();
buttonListener.click = function(){
  mx.managers.PopUpManager.createPopUp(_root, mx.containers.Window, true, {
  title:"Change Password", contentPath:"PasswordForm" })
}
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.

## 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()` or any applicable properties and methods of the Window class. For more information, see Window class.

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:

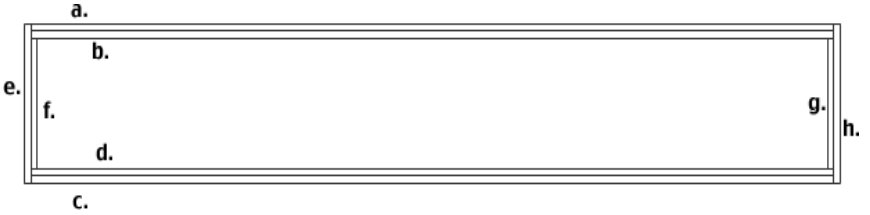
| Style | Description |
|---|---|
| borderStyle | The component border; either "none", "inset", "outset", or "solid". This style does not inherit its value. |

### 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 |
|---|
| `borderColor` |
| `highlightColor` |
| `borderColor` |
| `shadowColor` |
| `borderCapColor` |
| `shadowCapColor` |
| `shadowCapColor` |
| `borderCapColor` |

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 Namespace** 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` | Triggered when the close button is released. |
| `Window.mouseDownOutside` | Triggered when the mouse is pressed outside the modal window. |

Inherits all events from UIObject, UIComponent, View, and ScrollView.

## Window.click

### Availability

Flash Player 6.

### 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 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 249.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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 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 249.

### 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.

### 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.

#### 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 28.

## XMLConnector component

For the latest information about this feature, click the Update button at the top of the Help tab.

## XUpdateResolver component

For the latest information about this feature, click the Update button at the top of the Help tab.

# INDEX

## D



## E



## F

## G



## H



## I



## L



## M



## N

## T



## U



## V