8

Trademarks

1 Step RoboPDF, ActiveEdit, ActiveTest, Authorware, Blue Sky Software, Blue Sky, Breeze, Breezo, Captivate, Central, ColdFusion, Contribute, Database Explorer, Director, Dreamweaver, Fireworks, Flash, FlashCast, FlashHelp, Flash Lite, FlashPaper, Flash Video Encoder, Flex, Flex Builder, Fontographer, FreeHand, Generator, HomeSite, JRun, MacRecorder, Macromedia, MXML, RoboEngine, RoboHelp, RoboInfo, RoboPDF, Roundtrip, Roundtrip HTML, Shockwave, SoundEdit, Studio MX, UltraDev, and WebHelp are either registered trademarks or trademarks of Macromedia, Inc. and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words, or phrases mentioned within this publication may be trademarks, service marks, or trade names of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

Third-Party Information

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

Speech compression and decompression technology licensed from Nellymoser, Inc. (www.nellymoser.com).

Sorenson™ Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Opera ® browser Copyright © 1995-2002 Opera Software ASA and its suppliers. All rights reserved.

Macromedia Flash 8 video is powered by On2 TrueMotion video technology. © 1992-2005 On2 Technologies, Inc. All Rights Reserved. http://www.on2.com.

Visual SourceSafe is a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Copyright © 2005 Macromedia, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without written approval from Macromedia, Inc. Notwithstanding the foregoing, the owner or authorized user of a valid copy of the software with which this manual was provided may print out one copy of this manual from an electronic version of this manual for the sole purpose of such owner or authorized user learning to use such software, provided that no part of this manual may be printed out, reproduced, distributed, resold, or transmitted for any other purposes, including, without limitation, commercial purposes, such as selling copies of this documentation or providing paid-for support services.

Acknowledgments

Project Management: Sheila McGinn

Writing: Bob Berry, Jen deHaan, Peter deHaan, David Jacowitz, Wade Pickett

Managing Editor: Rosana Francescato

Lead Editor: Lisa Stanziano

Editing: Mary Ferguson, Mary Kraemer, Lisa Stanziano

Production Management: Patrice O’Neill, Kristin Conradi, Yuko Yagi

Media Design and Production: Adam Barnett, Aaron Begley, Paul Benkman. John Francis, Geeta Karmarkar, Masayo Noda, Paul Rangel, Arena Reed, Mario Reynoso

Special thanks to Jody Bleyle, Mary Burger, Lisa Friendly, Stephanie Gowin, Bonnie Loo, Nivesh Rajbhandari, Mary Ann Walsh, Erick Vera, the beta testers, and the entire Flash and Flash Player engineering and QA teams.

First Edition: September 2005

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

Contents

Intended audience 8
System requirements. 8
About the documentation 8
Typographical conventions 9
Terms used in this manual 9
Additional resources 9

Chapter 1: About Components. 11

Installing components 12
Where component files are stored 14
Modifying the component files 15
Benefits of using components. 16
Component categories 16
About version 2 component architecture 17
Version 2 component features 18
About compiled clips and SWC files 19
Accessibility and components. 20

Chapter 2: Creating an Application with Components (Flash Professional Only) 21

About the Fix Your Mistake tutorial 21
Build the main page 23
Bind data components to display gift ideas 29
Display gift details 33
Create the checkout screen 39
Test the application 47
Viewing the completed application 47

Chapter 3: Working with Components 49

The Components panel 50
Adding components to Flash documents 50
Components in the Library panel 54
Setting component parameters 55
Sizing components 56
Deleting components from Flash documents 57
Using code hints 58
Creating custom focus navigation 58
Managing component depth in a document 59
Components in Live Preview 60
Using a preloader with components 60
About loading components 62
Upgrading version 1 components to version 2 architecture 62

Chapter 4: Handling Component Events 63

Using listeners to handle events 64
Delegating events 73
About the event object 77
Using the on() event handler 78

Chapter 5: Customizing Components 81

Using styles to customize component color and text 82
About skinning components 96
About themes 108
Combining skinning and styles to customize a component 118

Chapter 6: Creating Components 125

Component source files 125
Overview of component structure 126
Building your first component 127
Selecting a parent class 136
Creating a component movie clip 138
Creating the ActionScript class file 143
Incorporating existing components within your component 173
Exporting and distributing a component 182
Final steps in component development 185

Chapter 7: Collection Properties 187
Defining a collection property 188
Simple collection example 189
Defining the class for a collection item 191
Accessing collection information programmatically. 191
Exporting components that have collections to SWC files. 194
Using a component that has a collection property 194

Index 197

Introduction

Macromedia Flash Basic 8 and Macromedia Flash Professional 8 are the standard authoring tools for producing high-impact web experiences. Components are the building blocks for the Rich Internet Applications that provide these experiences. A *component* is a movie clip with parameters that are set during authoring in Macromedia Flash, and with ActionScript methods, properties, and events that allow you to customize the component at runtime. Components are designed to allow developers to reuse and share code, and to encapsulate complex functionality that designers can use and customize without using ActionScript.

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

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

This chapter contains the following sections:

Intended audience 8
System requirements. 8
About the documentation 8
Typographical conventions 9
Terms used in this manual 9
Additional resources 9

Intended audience

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

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

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

For more information, see Chapter 3, “Working with Components,” on page 49.

System requirements

Macromedia components do not have any system requirements in addition to Flash.

Any SWF file that uses version 2 components must be viewed with Flash Player 6 (6.0.79.0) or later, and must be published for ActionScript 2.0 (you can set this through File > Publish Settings, in the Flash tab).

About the documentation

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

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

For information about Macromedia Flash, see the following documents:

- *Using Flash*
- *Learning ActionScript 2.0 in Flash*
- *ActionScript 2.0 Language Reference*
- *Components Language Reference*

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, including method and property names.
- *`Code font italic`* indicates a code item that should be replaced (for example, an ActionScript parameter).
- **Bold font** indicates a value that you enter.

Terms used in this manual

The following terms are used in this manual:

at runtime When the code is running in Flash Player.

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

Additional resources

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

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

For detailed information on ActionScript terms, syntax, and usage, see *Learning ActionScript 2.0 in Flash* and the *ActionScript 2.0 Language Reference*.

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

CHAPTER 1

About Components

1

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

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

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

NOTE: When publishing version 2 components, you must set your publish settings to publish for ActionScript 2.0 (File > Publish Settings, Flash tab). The version 2 components will not run correctly if published using ActionScript 1.0.

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.

For a complete list of components included with Flash Basic 8 and Flash Professional 8, see "Installing components" on page 12. You can also download components built by members of the Flash community at the Macromedia Exchange at www.macromedia.com/cfusion/exchange/index.cfm.

This chapter contains the following sections:

Installing components 12
Where component files are stored 14
Modifying the component files 15
Benefits of using components. 16
Component categories 16
About version 2 component architecture 17
Version 2 component features 18
About compiled clips and SWC files 19
Accessibility and components. 20

Installing components

A set of Macromedia components is already installed when you start Flash for the first time. You can view them in the Components panel.

Flash Basic 8 includes the following components:

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

Flash Professional 8 includes the Flash Basic 8 components plus the following additional components and classes:

- Accordion component (Flash Professional only)
- Alert component (Flash Professional only)
- Data binding classes (Flash Professional only)
- DateField component (Flash Professional only)
- DataGrid component (Flash Professional only)
- DataHolder component (Flash Professional only)
- DataSet component (Flash Professional only)
- DateChooser component (Flash Professional only)
- FLVPlayback Component (Flash Professional Only)
- Form class (Flash Professional only)
- Media components (Flash Professional only)
- Menu component (Flash Professional only)
- MenuBar component (Flash Professional only)
- RDBMSResolver component (Flash Professional only)
- Screen class (Flash Professional only)
- Slide class (Flash Professional only)
- Tree component (Flash Professional only)
- WebServiceConnector component (Flash Professional only)
- XMLConnector component (Flash Professional only)
- XUpdateResolver component (Flash Professional only)

To view the Flash Basic 8 or Flash Professional 8 components:

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

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

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

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

1. Quit Flash.
2. Place the SWC or FLA file containing the component in the following folder on your hard disk:
   - In Windows: C:\Program Files\Macromedia\Flash 8*language*\Configuration\Components
   - On the Macintosh: Macintosh HD/Applications/Macromedia Flash 8/Configuration/Components (Macintosh)
3. Start Flash.
4. Select Window > Components to view the component in the Components panel if it isn't already open.

Where component files are stored

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

> NOTE: For information about these folders, see "Configuration folders installed with Flash" in *Getting Started with Flash*.

Components are installed in the following locations:

- Windows 2000 or Windows XP: C:\Program Files\Macromedia\Flash 8*language*\Configuration\Components
- Mac OS X: Macintosh HD/Applications/Macromedia Flash 8/Configuration/Components

Modifying the component files

The source ActionScript files for components are located in:

- Windows 2000 or Windows XP: C:\Program Files\Macromedia\Flash 8*language*\First Run\Classes\mx
- Mac OS X: Macintosh HD/Applications/Macromedia Flash 8/First Run/Classes/mx

The files in the First Run directory are copied to your Documents and Settings path when Flash is first lunched. The Documents and Settings paths are:

- Windows 2000 or Windows XP: C:\Documents and Settings*username*\Local settings\Application Data\Macromedia\Flash 8*language*\Configuration\Classes\mx
- Mac OS X: *Username*/Library/Application Support/Macromedia/Flash 8/*language*/Configuration/Classes/mx

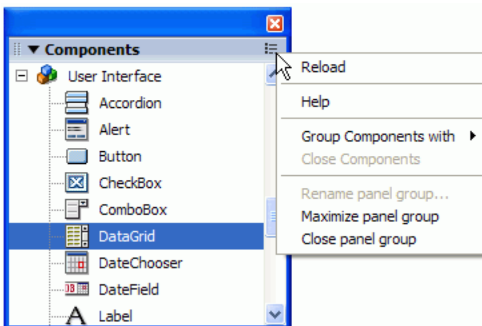
When Flash starts, if a file is missing from the Document and Settings path, Flash copies it over from the First Run directory to your Documents and Settings path.

> **NOTE** If you want to modify the source ActionScript files, modify the ones in the Documents and Settings path. If any of your modifications "break" a component, Flash will restore the original functionality when you close and relaunch Flash by copying the functional file from the First Run directory. However if you modify the files in the First run directory and that "breaks" a component, then you may need to reinstall Flash to restore the source files back to the functional ones.

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

To refresh the contents of the Components panel:

- Select Reload from the Components panel menu.



To remove a component from the Components panel:

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

Benefits of using components

Components enable you to separate the process of designing your application from the process of coding. They also let you to reuse code, either in components that 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.

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

Components that are based on the version 2 architecture share core functionality such as styles, event handling, skinning, focus management, and depth management. When you add the first version 2 component to an application, there is approximately 25K added to the document that provides this core functionality. When you add additional components, that same 25K is reused for them as well, resulting in a smaller increase in size to your document than you may expect. For information about upgrading components, see "Upgrading version 1 components to version 2 architecture" on page 62.

Component categories

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

- Data components (mx.data.*)

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

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

- FLVPlayback component (mx.video.FLVPlayback)

  The FLVPlayback component lets you readily include a video player in your Flash application to play progressive streaming video over HTTP, from a Flash Video Streaming Service (FVSS), or from Flash Communication Server (FCS).

- Media components (mx.controls.*)

  Media components let you play back and control streaming media; MediaController, MediaPlayback, and MediaDisplay are media components.
- User interface components (mx.controls.*)

  User interface components (often referred to as "UI Components") allow you to interact with an application; for example, the RadioButton, CheckBox, and TextInput components are user interface controls.
- Managers (mx.managers.*)

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

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

For a complete list of components, see *Components Language Reference*.

About version 2 component architecture

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

Flash components are built with version 2 of the Macromedia Component Architecture. Version 2 components are supported by Flash Player 6 (6.0.79.0) and later, and ActionScript 2.0. These components are not always compatible with components built using version 1 architecture (all components released before Flash MX 2004). Also, the original version 1 components are not supported by Flash Player 7. For more information, see "Upgrading version 1 components to version 2 architecture" on page 62.

> **NOTE** Flash MX UI components have been updated to work with Flash Player 7 or later. These updated components are still based on version 1 architecture. You can download them from the Macromedia Flash Exchange at www.macromedia.com/go/v1_components.

Version 2 components are included in the Components panel as compiled clip (SWC) symbols. A compiled clip is a component movie clip whose code has been compiled. Compiled clips cannot be edited, but you can change their parameters in the Property inspector and Component inspector, just as you would with any component. For more information, see "About compiled clips and SWC files" on page 19.

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

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

NOTE	The class hierarchy is also available as a FlashPaper file in the installation location: Flash 8\Samples and Tutorials\Samples\Components\arch_diagram.swf.

All components also use the same event model, CSS-based styles, and built-in themes and skinning mechanisms. For more information on styles and skinning, see Chapter 5, "Customizing Components," on page 81. For more information on event handling, see Chapter 3, "Working with Components," on page 49.

For a detailed explanation of the version 2 component architecture, see Chapter 6, "Creating Components," on page 125.

Version 2 component features

This section outlines the features of version 2 components (compared to version 1 components) from the perspective of a developer using components to build Flash applications. For detailed information about the differences between the version 1 and version 2 architectures for building components, see Chapter 6, "Creating Components," on page 125.

The Component inspector allows you to change component parameters while authoring in Macromedia Flash and Macromedia Dreamweaver. (See "Setting component parameters" on page 55.)

The listener event model allows listeners to handle events. (See Chapter 4, "Handling Component Events," on page 63.) Flash doesn't have a `clickHandler` parameter in the Property inspector, as there was in Flash MX; you must write ActionScript code to handle events.

Skin properties let you load individual skins (for example, up and down arrows or the check for a check box) at runtime. (See "About skinning components" on page 96.)

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 82.)

Themes allow you to drag a predesigned appearance from the library onto a set of components. (See “About themes” on page 108.)

The Halo theme is the default theme that the version 2 components use. (See “About themes” on page 108.)

Manager classes provide an easy way to handle focus and depth in a application. (See “Creating custom focus navigation” on page 58 and “Managing component depth in a document” on page 59.)

The base classes UIObject and UIComponent provide core methods, properties, and events to components that extend them. (See “UIComponent class” and “UIObject class” in the *Components Language Reference*.)

Packaging as a SWC file allows easy distribution and concealable code. See Chapter 6, “Creating Components,” on page 125.

Built-in data binding is available through the Component inspector. For more information, see “Data Integration (Flash Professional Only)” in *Using Flash*.

An easily extendable class hierarchy using ActionScript 2.0 allows you to create unique namespaces, import classes as needed, and subclass easily to extend components. See Chapter 6, “Creating Components,” on page 125 and the *ActionScript 2.0 Language Reference.*

NOTE: Flash 8 has several features that are not supported by the v2 components, including 9-slice (sometimes referred to as “scale-9”), FlashType, and bitmap caching.

About compiled clips and SWC files

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

Components included with Flash are not FLA files—they are compiled clips (that have been packaged into compiled clip (SWC) files. SWC is the Macromedia file format for distributing components; it contains a compiled clip, the component’s ActionScript class file, and other files that describe the component. For details about SWC files, see “Exporting and distributing a component” on page 182.

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.

To compile a movie clip:

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

To export a SWC file:

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

> **NOTE** Flash Basic 8 and Flash Professional 8 continue to support FLA components.

Accessibility and components

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

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

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

Most components built by Macromedia are also navigable by the keyboard. Each component's entry in the *Components Language Reference* indicates whether you can control the component with the keyboard.

CHAPTER 2

Creating an Application with Components (Flash Professional Only)

2

Components are prebuilt Flash elements that you can use when creating Macromedia Flash applications. Components include user interface controls, data access and connectivity mechanisms, and media-related elements. Components save you work when building a Flash application by providing you with elements and behavior that you would need to create from scratch otherwise.

This chapter contains a tutorial that shows you how to build a Flash application using components that are available in Macromedia Flash Professional 8. You will learn how to work with components in the Flash authoring environment and also learn how to make them interactive with ActionScript code.

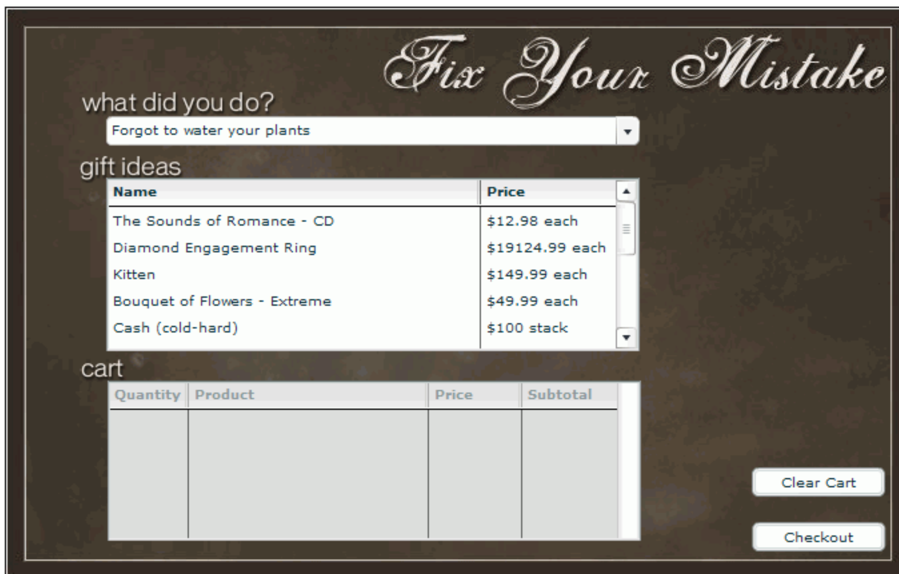
About the Fix Your Mistake tutorial

This tutorial takes you through the steps to create a basic online shopping application for the "Fix Your Mistake" gift service. This service helps a user select an appropriate gift to make amends to someone whom the user has offended. The application filters a list of gifts to those choices that match the severity of the user's offense. From that list the user can add items to the shopping cart and then proceed to the checkout page to provide billing, shipping, and credit card information.

This chapter contains the following sections:

About the Fix Your Mistake tutorial 21
Build the main page 23
Bind data components to display gift ideas 29
Display gift details 33
Create the checkout screen 39
Test the application 47
Viewing the completed application 47

The application uses the ComboBox, DataGrid, TextArea, and Button components, as well as others, to create the application interface. The main page of the interface looks like this:



The application uses the ActionScript WebService class to connect dynamically to a web service to retrieve the list of offenses (problems.xml) that appear in the combo box. It also uses ActionScript to handle the user's interactions with the application.

The application uses data components to connect the interface to another data source. It uses the XMLConnector component to connect to an XML data file (products.xml) for the list of gifts and it uses the DataSet component to filter the data and present it to the data grid.

The tutorial requires some familiarity with the Flash authoring environment and some experience with ActionScript. In the authoring environment, you should have some experience using panels, tools, the timeline, and the library. All the ActionScript needed for creating the sample application is provided here within the tutorial. To understand the scripting concepts and create your own applications, however, you will need additional experience writing ActionScript.

To see a working version of the completed application, see "Viewing the completed application" on page 47

Keep in mind that the sample application is for demonstration purposes and therefore is not as complete as a real-world application.

Build the main page

Follow these steps to create the application's main page by adding components to a skeletal starter page. Then add ActionScript code to customize the components, import the ActionScript classes that allow you to manipulate the application's components, and access a web service to populate the combo box with a list of offenses. The code populates the combo box by setting its `dataProvider` property to receive the results from the web service.

1. Open the first_app_start.fla file, which you can find at one of the following locations:
   - In Windows: *install drive:*\Program Files\Macromedia\Flash 8\Samples and Tutorials\Samples\Components\ComponentsApplication
   - On the Macintosh: *Macintosh HD*/Applications/Macromedia Flash 8/Samples and Tutorials/Samples/Components/ComponentsApplication

   The file contains a start page that looks like the following:

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

4. Make sure the Form layer is selected. In the Components panel (Window > Components), locate the ComboBox component in the User Interface tree. Drag an instance of ComboBox onto the Stage. Place it below the What Did You Do? text. In the Property inspector (Window > Properties > Properties), enter **problems_cb** for the instance name. Enter **400** (pixels) for the width. Enter **76.0** for the *x* location and **82.0** for the *y* location.

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

5. Drag an instance of the DataGrid component from the User Interface tree in the Components panel onto the Stage. Place it below the Gift Ideas text. Enter **products_dg** for the instance name. Enter **400** (pixels) for the width and **130** for the height. Enter **76.0** for the *x* location and **128.0** for the *y* location.
6. Drag an instance of the DataSet component from the Data tree in the Components panel onto the side of the Stage. (The DataSet component does not appear in the application at runtime. The DataSet icon is simply a placeholder that you work with in the Flash authoring environment.) Enter **products_ds** for the instance name.

   Drag an instance of the XMLConnector component from the Data tree in the Components panel to the side of the Stage. (Like the DataSet component, the XMLConnector component does not appear in the application at runtime.) Enter **products_xmlcon** for the instance name. Click the Parameters tab in the Property inspector, and enter www.flash-mx.com/mm/firstapp/products.xml for the `URL` property. Click the value for the direction property to activate the combo box, click the down-arrow and select `receive` from the list.

NOTE	You can also use the Component inspector (Window > Component Inspector) to set parameters for components. The Parameters tab in the Property inspector and the Component inspector work in the same way.

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

7. Drag an instance of the Button component from the User Interface tree in the Components panel onto the Stage. Place it in the lower-right corner of the Stage. Enter **checkout_button** for the instance name. Click the Parameters tab and enter **Checkout** for the `label` property. For the *x* and *y* coordinates, enter **560.3** and **386.0**, respectively.

Import the component classes

Each component is associated with an ActionScript class file that defines its methods and properties. In this section of the tutorial, you will add ActionScript code to import the classes associated with the application's components. For some of these components, you have already added instances to the Stage. For others, you will add ActionScript later in the tutorial to create instances dynamically.

The import statement creates a reference to the class name and makes it easier to write ActionScript for the component. The import statement enables you to refer to the class by its class name rather than its complete name, which includes the package name. For example, after you create a reference to the ComboBox class file with an import statement, you can refer to instances of the combo box with the syntax `instanceName:ComboBox`, rather than `instanceName:mx.controls.ComboBox`.

A *package* is a directory that contains class files and resides in a designated classpath directory. You can use a wild card character to create references to all the classes in a package: for example, the syntax `mx.controls.*` creates references to all classes in the controls package. (When you create a reference to a package with a wild card, the unused classes are dropped from the application when it is compiled, so they don't add any extra size.)

For the application in this tutorial, you need the following packages and individual classes:

UI Components Controls package This package contains classes for the user interface control components, including ComboBox, DataGrid, Loader, TextInput, Label, NumericStepper, Button, and CheckBox.

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

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

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

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

To import these classes, you will create an Actions layer and add the ActionScript code to the first frame of the main timeline. All the code that you will add to the application in the remaining steps of the tutorial should be placed in the Actions layer.

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

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

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

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

Set the data types of component instances

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

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

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

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

> **NOTE** The instance names you specify here must agree with the instance names that you assigned when you dragged the components to the Stage.

Customize the appearance of components

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

- Add the following code to set styles:

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

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

Display offenses in the combo box

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

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

- In the Actions panel, add the following code:

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

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

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

> **TIP** Press Control+S to save your work and then Control+Enter (or select Control > Test Movie) to test the application. The combo box should be populated with a list of offenses at this point and you should see the empty data grid that you created for Gift Ideas, along with the checkout button.

Bind data components to display gift ideas

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

Now you will use data binding features in the Flash authoring environment to bind the XMLConnector, DataSet, and DataGrid components together to use the XML data in the application. For general information on working with data binding and other features of the Flash data integration architecture, see Chapter 16, “Data Integration (Flash Professional Only)” in *Using Flash*.

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

Use schema to describe the XML data source

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

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

In the top pane of the Schema tab, select the `image` element. In the bottom pane, select `data type` and change the value from <empty> to String. Repeat this step for the `description` element.

Filter the gift ideas to match the offense

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

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

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

   This option means that the data will pass from the `products_xmlcon` instance to the `products_ds` instance (rather than passing in both directions, or passing from the DataSet instance to the XMLConnector instance).
8. On the Stage, select the `products_ds` instance. In the Bindings tab of the Component inspector, notice that the component's data provider appears in the Binding List (the top pane of the Bindings tab). In the Binding Attributes pane, the Bound To parameter indicates that the `products_ds` instance is bound to the `products_xmlcom` instance, and the binding direction is In.

   In the next few steps you will bind the DataSet instance to the DataGrid instance so that the data that is filtered by the data set will be displayed in the data grid.
9. With the `products_ds` instance still selected, click the Add Binding button in the Bindings tab.
10. In the Add Binding dialog box, select the `dataProvider: array` item and click OK.
11. In the Bindings tab, make sure the `dataProvider: array` item is selected in the Binding List.
12. Click the Bound To item in the Binding Attributes pane.

13. In the Value column for the Bound To item, click the magnifying glass icon to open the Bound To dialog box.
14. In the Bound To dialog box, select the `products_dg` (DataGrid) instance in the Component Path pane. Select `dataProvider:array` in the Schema Location pane. Click OK.

Add columns to the Gift Ideas section

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

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

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

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

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

Trigger the XML Connector

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

- Add the following code in the Actions panel:

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

Add an event listener to filter the gift ideas

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

For more information on working with event listeners, see "Using event listeners" in *Learning ActionScript 2.0 in Flash*.

■ In the Actions panel, add the following code:

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

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

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

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

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

Add the cart

The next code that you will add creates an instance of the custom Cart class and initializes it.

- In the Actions panel, add the following code:

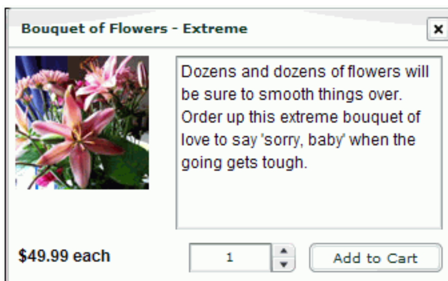
```
var myCart:Cart = new Cart(this);
myCart.init();
```

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

TIP	Press Control+S to save your work and then Control+Enter (or select Control->Test Movie) to test the application. When you select an offense in the combo box, the data grid that you created for Gift Ideas should display a subset of gifts to match the selected offense.

Display gift details

A pop-up window appears in the application when a user clicks a product in the Gift Ideas section. The pop-up window contains component instances that display information about the product, including a text description, an image, and the price. To make this pop-up window, you will create a movie clip symbol and add instances of the Loader, TextArea, Label, NumericStepper, and Button components. The product detail window for Bouquet of Flowers Extreme looks like this:



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

1. Drag an instance of the Window component from the User Interface tree in the Components panel to the library.

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

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

   For movie clip symbols that are in the library but not on the Stage, you must select Export for ActionScript so that you can manipulate them using ActionScript. (Exporting in first frame means that the movie clip is available as soon as the first frame loads.) Later in the tutorial you will add ActionScript that will generate an instance of the movie clip dynamically each time a user clicks a product in the Gift Ideas section.

5. In the Timeline for the new symbol, select Layer 1 and rename it **Components**.
6. Drag an instance of the Loader component from the User Interface tree in the Components panel onto the Stage. Enter **5**, **5** for the *x*, *y* coordinates respectively. Enter **image_ldr** for the instance name. Click the Parameters tab in the Property inspector. Select `false` for `autoLoad` and `false` for `scaleContent`.

   The Loader component instance will be used to display an image of the product. The `false` setting for `autoLoad` specifies that the image will not load automatically. The `false` setting for `scaleContent` specifies that the image will not be scaled. Later in the tutorial you will add code that loads the image dynamically, based on the product that the user selects in the Gift Ideas section.

7. Drag an instance of the TextArea component from the User Interface tree in the Components panel onto the Stage. Place it next to the Loader component. Enter **125**, **5** for the *x*, *y* coordinates respectively. Enter **description_ta** for the instance name. Set the Width to **200** and Height to **130**. Click the Parameters tab in the Property inspector. For editable, select `false`. For `html`, select `true`. For `wordWrap`, select `true`.

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

8. Drag an instance of the Label component from the User Interface tree in the Components panel onto the Stage. Place it below the Loader component. Set the *x*, *y* coordinates to **5**, **145**. Enter **price_lbl** for the instance name. Click the Parameters tab in the Property inspector. For `autoSize`, select `left`. For `html`, select `true`.

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

9. Drag an instance of the NumericStepper component from the User Interface tree in the Components panel onto the Stage. Place it below the TextArea component. Set the *x*, *y* coordinates to **135**, **145**. Enter **quantity_ns** for the instance name. Click the Parameters tab in the Property inspector. For `minimum`, enter **1**.

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

10. Drag an instance of the Button component from the User Interface tree in the Components panel onto the Stage. Place it beside the NumericStepper component. Set the *x*, *y* coordinates to **225**, **145**. Enter **addToCart_button** for the instance name. Click the Parameters tab in the Property inspector. For `label`, enter **Add To Cart**.

Add an event listener to trigger the display of gift details

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

- In the Actions panel of the main Timeline, add the following code:

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

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

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

Add code to the ProductForm movie clip

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

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

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

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

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

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

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

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

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

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

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

   You should check syntax frequently as you add code to an application. Any errors found in the code are listed in the Output panel. (When you check syntax, only the current script is checked; other scripts that may be in the FLA file are not checked.) For more information, see “Debugging your scripts” in *Learning ActionScript 2.0 in Flash*.

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

> **TIP** Press Control+S to save your work and then Control+Enter (or select Control ›Test Movie) to test your application. When you click a gift selection now, a window should appear and display an image of the gift, accompanied by a description, the price, and a numeric stepper that allows you to choose the quantity that you want.

Create the checkout screen

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

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

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

1. In the main Timeline for the application, move the playhead to Frame 10 (labeled Checkout). Make sure the Form layer is selected.
2. Insert a blank keyframe on Frame 10 in the Form layer (select the frame and select Insert > Timeline > Blank Keyframe).
3. With the new keyframe selected, drag an instance of the Accordion component from the User Interface tree in the Components panel onto the Stage. In the Property inspector, enter **checkout_acc** for the instance name. Set the width to **300** pixels and the height to **200** pixels.

4. Drag an instance of the Button component from the User Interface tree in the Components panel onto the lower-right corner of the Stage. In the Property inspector, enter **back_button** for the instance name. Click the Parameters tab, and enter **Back** for the `label` property.

About the Billing, Shipping, and Credit Card panes

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

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

Create the Billing Information pane

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

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

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

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

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

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

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

13. Select Frame 1 of the new layer.
14. In the Tools panel, select the Rectangle tool. Set the Stroke color to None and the Fill color to white.

    Click the Stroke Color control in the Tools panel and click the None button—the white swatch with a red line through it. Click the Fill Color control and click the white color swatch.
15. Drag to create a rectangle that extends beyond the bottom and right edges of the Label and TextInput instances.

Create the Shipping Information pane

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

1. Follow the earlier instructions (see "Create the Billing Information pane" on page 40) to create the movie clips for the Credit Card Information pane. Note these naming differences:
   - For the first movie clip, enter **checkout2_mc** for the symbol name and **checkout2_sp** for the instance name. In the Property inspector's Parameters tab, set the `contentPath` property to **checkout2_sub_mc**.
   - For the second movie clip, enter **checkout2_sub_mc** for the symbol name.
   - For the TextInput instances, change "billing" to "shipping" in the instance names.
2. With the `checkout2_sub_mc` movie clip open in symbol-editing mode, drag an instance of the CheckBox component onto the Stage and position it just above the first Label instance.

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

Create the Credit Card Information pane

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

1. Follow steps 1-9 of the Billing Information instructions (see "Create the Billing Information pane" on page 40) to create the movie clips for the Credit Card Information pane. Note these naming differences:
   - For the first movie clip, enter **checkout3_mc** for the symbol name and **checkout3_sp** for the instance name. In the Property inspector's Parameters tab, set the `contentPath` property to **checkout3_sub_mc**.
   - For the second movie clip, enter **checkout3_sub_mc** for the symbol name.
2. Drag four instances of the Label component onto the Stage. Name and position the instances as follows:
   - For the first instance, enter **ccName_lbl** for the instance name and set the *x* and *y* coordinates to **5**, **5**. Click the Parameters tab and enter **Name On Card** for `text`.
   - For the second instance, enter **ccType_lbl** for the instance name and set the *x* and *y* coordinates to **5**, **35**. Click the Parameters tab and enter **Card Type** for `text`.
   - For the third instance, enter **ccNumber_lbl** for the instance name and set the *x* and *y* coordinates to **5**, **65**. Click the Parameters tab and enter **Card Number** for `text`.
   - For the fourth instance, enter **ccExp_lbl** for the instance name and set the *x* and *y* coordinates to **5**, **95**. Click the Parameters tab and enter **Expiration** for `text`.
3. Drag an instance of the TextInput component onto the Stage and position it to the right of the `ccName_lbl` instance. Name the new instance **ccName_ti**. Set the *x* and *y* coordinates to **105**, **5**. Set the width to **140**.
4. Drag an instance of the ComboBox component onto the Stage and position it to the right of the `ccType_lbl` instance. Name the new instance **ccType_cb**. Set the *x* and *y* coordinates to **105**, **35**. Set the width to **140**.
5. Drag another instance of the TextInput component onto the Stage and position it to the right of the `ccNumber_lbl` instance. Name the new instance **ccNumber_ti**. Set the *x* and *y* coordinates to **105**, **65**. Set Width to **140**.
6. Drag two instances of the ComboBox component onto the Stage. Position one to the right of the `ccExp_lbl` instance, and position the other one to the right of that. Name the first new instance **ccMonth_cb**. Set Width to **60** and the *x* and *y* coordinates to **105**, **95**. Name the second **ccYear_cb**. Set Width to **70** and the *x* and *y* coordinates to **175**, **95**.

7. Drag an instance of the Button component onto the Stage and position it at the bottom of the form, below the ccMonth_cb instance. Name the new instance **checkout_button**. Set the *x* and *y* coordinates to **125**, **135**. In the Property inspector's Parameters tab, set the `label` property to **Checkout**.
8. Follow the instructions in steps 14-15 of the Billing Information instructions (see "Create the Billing Information pane" on page 40) to add a rectangle to the bottom of the form.

Add an event listener to the Checkout button

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

- In the Actions panel for the main page, add the following code:

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

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

Add code for the Checkout screen

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

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

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

// Define the Accordion component on the Stage.
var checkout_acc:Accordion;
```

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

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

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

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

```
/* Add the second child to the Accordion.
Add an event listener for the sameAsBilling_ch CheckBox.
This copies the form values from the first child into the second child.
  */
var child2 = checkout_acc.createChild("checkout2_mc", "child2_mc",
  {label:"2. Shipping Information"});
var thisChild2 = child2.checkout2_sp.spContentHolder;
var checkboxListener:Object = new Object();
checkboxListener.click = function(evt:Object) {
  if (evt.target.selected) {
    thisChild2.shippingFirstName_ti.text =
  thisChild1.billingFirstName_ti.text;
    thisChild2.shippingLastName_ti.text =
  thisChild1.billingLastName_ti.text;
    thisChild2.shippingCountry_ti.text =
  thisChild1.billingCountry_ti.text;
    thisChild2.shippingProvince_ti.text =
  thisChild1.billingProvince_ti.text;
    thisChild2.shippingCity_ti.text = thisChild1.billingCity_ti.text;
    thisChild2.shippingPostal_ti.text =
  thisChild1.billingPostal_ti.text;
  }
};
thisChild2.sameAsBilling_ch.addEventListener("click", checkboxListener);
```

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

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

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

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

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

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

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

```
/* Create a listener for the checkout_button Button instance.
This listener sends all the form variables to the server when the user
  clicks the Checkout button. */
var checkoutListener:Object = new Object();
checkoutListener.click = function(evt:Object){
  evt.target.enabled = false;
  /* Create two LoadVars object instances, which send variables to
  and receive results from the remote server. */
  var response_lv:LoadVars = new LoadVars();
  var checkout_lv:LoadVars = new LoadVars();
  checkout_lv.billingFirstName = thisChild1.billingFirstName_ti.text;
  checkout_lv.billingLastName = thisChild1.billingLastName_ti.text;
  checkout_lv.billingCountry = thisChild1.billingCountry_ti.text;
  checkout_lv.billingProvince = thisChild1.billingProvince_ti.text;
  checkout_lv.billingCity = thisChild1.billingCity_ti.text;
  checkout_lv.billingPostal = thisChild1.billingPostal_ti.text;
  checkout_lv.shippingFirstName = thisChild2.shippingFirstName_ti.text;
  checkout_lv.shippingLastName = thisChild2.shippingLastName_ti.text;
  checkout_lv.shippingCountry = thisChild2.shippingCountry_ti.text;
  checkout_lv.shippingProvince = thisChild2.shippingProvince_ti.text;
  checkout_lv.shippingCity = thisChild2.shippingCity_ti.text;
  checkout_lv.shippingPostal = thisChild2.shippingPostal_ti.text;
  checkout_lv.ccName = thisChild3.ccName_ti.text;
  checkout_lv.ccType = thisChild3.ccType_cb.selectedItem;
  checkout_lv.ccNumber = thisChild3.ccNumber_ti.text;
```

```
    checkout_lv.ccMonth = thisChild3.ccMonth_cb.selectedItem;
    checkout_lv.ccYear = thisChild3.ccYear_cb.selectedItem;

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

Test the application

Congratulations! You've finished building the application. Now press Control+S to save your work and then Control+Enter (or select Control >Test Movie) to test the application.

Viewing the completed application

In the event that you have not been able to successfully complete the tutorial, you can view a working version of the completed application. You can find this starter Flash (FLA) file, first_app_start.fla, and the finished file, first_app.fla, in the Samples folder on your hard disk:

- In Windows: *boot drive*\Program Files\Macromedia\Flash 8\Samples and Tutorials\Samples\Components\ComponentsApplication.
- On the Macintosh: *Macintosh HD*/Applications/Macromedia Flash 8/Samples and Tutorials/Samples/Components/ComponentsApplication.

To view the FLA file for the application, open the first_app.fla file in the components_application folder.

You can compare these files to your own to help you find your errors.

All the components used in the application appear in the library (along with graphics files and other assets used to create the application). Some components appear as instances on the Stage. Some are referenced in the ActionScript code and do not appear until runtime.

CHAPTER 3

Working with Components

3

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

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

- (Windows) Program Files\Macromedia\Flash 8\Samples and Tutorials\Samples\Components\TipCalculator
- (Macintosh) Applications/Macromedia Flash 8/Samples and Tutorials/Samples/Components/TipCalculator

This chapter covers the following topics:

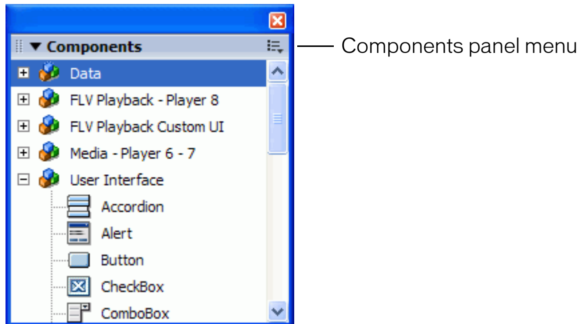
The Components panel 50
Adding components to Flash documents 50
Components in the Library panel 54
Setting component parameters 55
Sizing components 56
Deleting components from Flash documents 57
Using code hints 58
Creating custom focus navigation 58
Managing component depth in a document 59
Components in Live Preview 60
Using a preloader with components 60
About loading components 62
Upgrading version 1 components to version 2 architecture 62

The Components panel

All components in the user-level configuration/Components directory are displayed in the Components panel. (For more information about this directory, see "Where component files are stored" on page 14.)

To display the Components panel:

- Select Window > Components.



To display components that were installed after Flash starts:

1. Select Window > Components.
2. Select Reload from the Components panel pop-up menu.

Adding components to Flash documents

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

NOTE: The Menu and Alert components are two exceptions, and cannot be instantiated using `UIObject.createClassObject()`. They use the `show()` method instead.

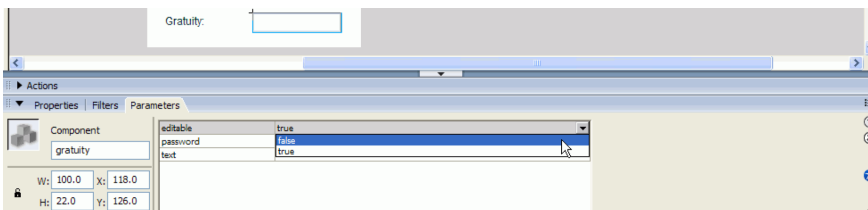
Adding components during authoring

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

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

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

   The following illustration shows the Property inspector for the TextInput component that is in the TipCalculator.fla sample file (installed at Flash 8/Samples and Tutorials/Samples/Components/TipCalculator).



For more information, see "Setting component parameters" on page 55.

8. Change the size of the component as desired by editing the values for the width and height.

   For more information on sizing specific component types, see the individual component entries in *Components Language Reference*.

9. If you want to change the color and text formatting of a component, do one or more of the following:
   - Set or change a specific style property value for a component instance by using the `setStyle()` method, which is available to all components. For more information, see `UIObject.setStyle() on page 1343`.
   - Edit multiple properties in the global style declaration assigned to all version 2 components.
   - Create a custom style declaration for specific component instances.

     For more information, see "Using styles to customize component color and text" on page 82.

10. If you want to customize the appearance of the component, do one of the following:
    - Apply a theme (see "About themes" on page 108).
    - Edit a component's skins (see "About skinning components" on page 96).

Adding components at runtime with ActionScript

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

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

Version 2 components that are installed with Flash reside in package directories. (For more information, see "About packages" in *Learning ActionScript 2.0 in Flash*. If you add a component to the Stage during authoring, you can refer to the component simply by using its instance name (for example, `myButton`). However, if you add a component to an application with ActionScript (at runtime), you must either specify its fully qualified class name (for example, `mx.controls.Button`) or import the package by using the `import` statement.

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

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

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

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

For more information, see "About importing class files" in *Learning ActionScript 2.0 in Flash*.

You can use ActionScript methods to set additional parameters for dynamically added components. For more information, see *Components Language Reference*.

> **NOTE**
> To add a component to a document at runtime, it must be in the library when the SWF file is compiled. To add a component to the library, drag the component icon from the Components panel to the library. Furthermore, if you are loading a movie clip containing a dynamically instantiated (using ActionScript) component into another movie clip, the parent movie clip must have the component in the library when the SWF file is compiled.

To add a component to your Flash document using ActionScript:

1. Drag a component from the Components panel into the library for the current document.

    > **NOTE**
    > Components are set to Export in First Frame by default (right-click for Windows, or control-click for Macintosh, and select the Linkage menu option to see the Export in First Frame setting). If you want to use a preloader for an application containing components, you need to change the export frame, see "Using a preloader with components" on page 60 for instructions.

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

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

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

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

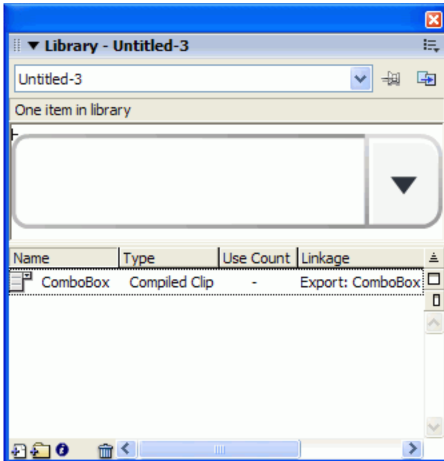
    Alternatively, you can import the class package, as in the following example:

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

    For more information, see UIObject.createClassObject() on page 1323 and Chapter 4, "Handling Component Events," on page 63.

Components in the Library panel

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



A ComboBox component in the Library panel

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

For more information about compiled clips, see "About compiled clips and SWC files" on page 19.

Setting component parameters

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

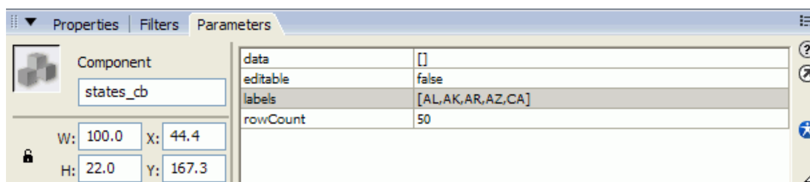
All version 2 User Interface (UI) components inherit properties and methods from the UIObject and UIComponent classes; these are the properties and methods that all components use, such as UIObject.setSize(), UIObject.setStyle(), UIObject.x, and UIObject.y. Each component also has unique properties and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

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

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

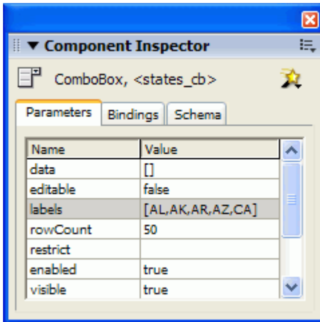
1. Select Window > Properties > Properties.
2. Select an instance of a component on the Stage.
3. Enter an instance name in the text box under the word *Component*.

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



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

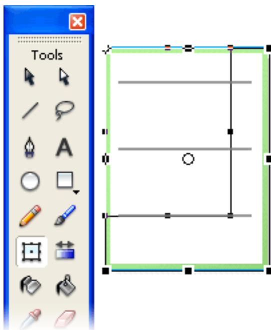
1. Select Window > Component Inspector.
2. Select an instance of a component on the Stage.
3. To enter parameters, click the Parameters tab.



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

Sizing components

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



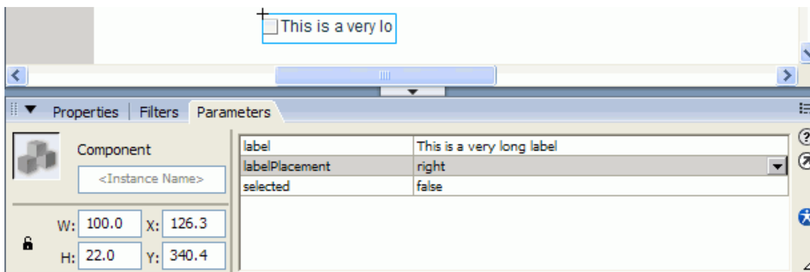
Resizing the Menu component on the Stage with the Free Transform tool

You can call the `setSize()` method from any component instance (see UIObject.setSize() on page 1341) to resize it. The following code resizes the TextArea component to 200 pixels wide and 300 pixels high:

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

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

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



A clipped label for the CheckBox component

For more information about sizing components, see their individual entries in the *Components Language Reference*.

Deleting components from Flash documents

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

To delete a component from a document:

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

Using code hints

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

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

As soon as you type the period after `myCheckBox`, Flash displays a list of methods and properties available for CheckBox components, because you have designated the variable as type CheckBox. For more information, see "About assigning data types and strict data typing" and "Using code hints" in *Learning ActionScript 2.0 in Flash*.

Creating custom focus navigation

When a user presses the Tab key to navigate in a Flash application or clicks in an application, the FocusManager class determines which component receives input focus (for more information see FocusManager class in the *Components Language Reference*). You don't need to add a FocusManager instance to an application or write any code to activate the Focus Manager.

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

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

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

```
var comment:mx.controls.TextArea;
var okButton:mx.controls.Button;
comment.tabIndex = 1;
okButton.tabIndex = 2;
```

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

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

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

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

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

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

For more information about creating a focus scheme in a Flash application, see FocusManager class in the *Components Language Reference*.

Managing component depth in a document

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

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

To use the Depth Manager, call its methods.

The following code places the component instance `loader` at a lower depth than the `button` component (and in the published SWF file it will appear “below” the button, if they overlap):

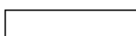
```
loader.setDepthBelow(button);
```

> **NOTE** You can also manage relative depths using Layers and the Modify > Arrange menu options within your document. Components adhere to the same rules for runtime depth management using layers and arrangement as do movie clips.

Components in Live Preview

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

A Button component with Live Preview enabled



A Button component with Live Preview disabled

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

To turn Live Preview on or off:

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

Using a preloader with components

Preloading involves loading some of the data for a SWF file before the user starts interacting with it. By default, components and classes are set to export in the first frame of the document that contains components. Because the components and classes are the first data to load, you might have problems implementing a progress bar or loading animation. Specifically, the components and classes might load before the progress bar, but you want the progress bar to reflect the loading progress of all data (including classes). Therefore, you should load the classes after other parts of the SWF file, but before you use components.

To do this, when you create a custom preloader for an application containing components, set the file's publish settings to export all the classes to the frame containing your components. To see a list of all the components in the Halo and Sample themes that have their assets set to Export in First Frame, see "Changing export settings" on page 116.

To change the export frame for all your classes:

1. Select File > Publish Settings.
2. In the Flash tab of the Publish Settings dialog box, make sure the ActionScript version is set to ActionScript 2.0.
3. Click the Settings button to the right of the ActionScript version.
4. In ActionScript 2.0 Settings, change the number for the Export Frame for Classes text box to the frame where your components first appear.

You cannot use any classes until the playhead reaches the frame you choose to load them into. Because components require classes for their functionality, you must load components after the frame specified for loading classes. If you export your classes to Frame 3, you cannot use anything from those classes until the playhead reaches Frame 3 and loads the data.

If you want to preload a file that uses components, you must also preload the components in the SWF file. To accomplish this, you must set your components to export for a different frame in the SWF file.

To change the frame into which components are exported:

1. Select Window > Library to open the Library panel.
2. Right-click (Windows) or Control-click (Macintosh) the component in the library.
3. Select Linkage from the context menu.
4. Deselect Export in First Frame.
5. Click OK.
6. Select File > Publish Settings.
7. Select the Flash tab and click the Settings button.
8. Enter a number into the Export Frame for Classes text box and click OK. The classes will load into this frame.
9. Click OK to close the Publish Settings dialog box.

If components do not load on the first frame, you can create a custom progress bar for the first frame of the SWF file. Do not reference any components in your ActionScript or include any components on the Stage until you load the classes for the frame you specified in Step 7.

> **NOTE** Components must be exported after the ActionScript classes that they use.

About loading components

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

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

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

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

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

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

The `_lockroot` property is supported by Flash Player 7. For information about this property, see _lockroot (MovieClip._lockroot property) in the *ActionScript 2.0 Language Reference.*

Upgrading version 1 components to version 2 architecture

The version 2 components were written to comply with several web standards (regarding events [www.w3.org/TR/DOM-Level-3-Events/events.html], styles, getter/setter policies, and so on) and are very different from their version 1 counterparts that were released with Macromedia Flash MX and in the DRKs that were released before Macromedia Flash MX 2004. Version 2 components have different APIs and were written in ActionScript 2.0. Therefore, using version 1 and version 2 components together in an application can cause unpredictable behavior. For information about upgrading version 1 components to use version 2 event handling, styles, and getter/setter access to the properties instead of methods, see Chapter 6, "Creating Components," on page 125.

Flash applications that contain version 1 components work properly in Flash Player 6 and Flash Player 7, when published for Flash Player 6 or Flash Player 6 (6.0.65.0). If you want to update your applications to work when published for Flash Player 7, you must convert your code to use strict data typing. For more information, see "Writing custom class files" in *Learning ActionScript 2.0 in Flash.*

CHAPTER 4

Handling Component Events

4

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

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

This chapter uses several versions of a simple Macromedia Flash application, TipCalculator, to teach you how to handle component events. The FLA and SWF files are installed with Flash to:

- In Windows: the C:\Program Files\Macromedia\Flash 8\Samples and Tutorials\Samples\Components\TipCalculator folder.
- On the Macintosh: HD/Applications/Macromedia Flash 8/Samples and Tutorials/Samples/Components/TipCalculator folder.

This chapter contains the following sections:

Using listeners to handle events 64
Delegating events. 73
About the event object 77
Using the on() event handler 78

Using listeners to handle events

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

- All events are broadcast by an instance of a component class. (The component instance is the *broadcaster.*)
- A *listener* can be a function or an object. If the listener is an object, it must have a callback function defined on it. The listener *handles* the event; this means the callback function is executed when the event occurs.
- To register a listener to a broadcaster, call the `addEventListener()` method from the broadcaster. Use the following syntax:

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

- You can register multiple listeners to one component instance.

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

- You can register one listener to multiple component instances.

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

- The handler function is passed an event object.

  You can use the event object in the body of the function to retrieve information about the event type, and the instance that broadcast the event. See "About the event object" on page 77.
- A listener object remains active until explicitly removed using `EventDispatcher.removeEventListener()`. For example:

  ```
  myComponent.removeEventListener("change", ListenerObj);
  ```

Using listener objects

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

- Use `this` in most situations.

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

- Use an existing object if it is convenient.

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

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

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

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

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

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

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

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

> **TIP** The *evtObj* parameter is an object that is automatically generated when an event is triggered and passed to the callback function. The event object has properties that contain information about the event. For details, see "About the event object" on page 77.

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

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

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

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

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

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

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

For information about the events a component broadcasts, see the component's entry in the *Components Language Reference*. For example, Button component events are listed in the Button component section (or Help > Components Language Reference > Button component > Button class > Event summary for the Button class).

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

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

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

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

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

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

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

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

1. Open the file TipCalculator.fla from the location specified in "Working with Components" on page 49.
2. Open the file TipCalculator.as from the location specified in "Working with Components" on page 49.
3. In the FLA file, select form1 and view the class name, TipCalculator, in the Property inspector.

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

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

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

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

Using the handleEvent callback function

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

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

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

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

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

Using listener functions

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

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

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

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

TIP The *`evtObj`* parameter is an object that is automatically generated when an event is triggered and passed to the function. The event object has properties that contain information about the event. For details, see "About the event object" on page 77.

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

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

You can call `addEventListener()` from any component instance; it is included in every UI component from the EventDispatcher class. For more information, see `EventDispatcher.addEventListener()`.

For information about the events a component broadcasts, see each component's entry in the *Components Language Reference*.

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

1. In Flash, select File > New and create a new Flash document.
2. Drag a List component to the Stage from the Components panel.
3. In the Property inspector, enter the instance name **myList**.
4. Select Frame 1 in the Timeline.
5. Select Window > Actions.
6. In the Actions panel, enter the following code:

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

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

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

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

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

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

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

> **CAUTION** In a listener function, the keyword `this` refers to the component instance that calls `addEventListener()`, not to the timeline or the class where the function is defined. However, you can use the Delegate class to delegate the listener function to a different scope. See "Delegating events" on page 73. To see an example of function scoping, see the next section.

About scope in listeners

Scope refers to the object within which a function executes. Any variable references within that function are recognized as properties of that object. You can use the Delegate class to specify the scope of a listener. For more information, see "Delegating events" on page 73.

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

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

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

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

However, if you pass `addEventListener()` a function, the function is invoked in the scope of the component instance that calls `addEventListener()`. This means that the keyword `this`, when used inside the function, refers to the broadcasting component instance. This causes a problem if you're defining the function in a class file. You cannot access the properties and methods of the class file with the expected paths because `this` doesn't point to an instance of the class. To work around this problem, use the Delegate class to delegate a function to the correct scope. See "Delegating events" on page 73.

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

```
class Cart extends MovieClip {

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

  function myHandler(eventObj:Object){

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

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

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

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

Delegating events

You can import the Delegate class into your scripts or classes to delegate events to specific scopes and functions (see "Delegate class" in the *Components Language Reference*). To import the Delegate class, use the following syntax:

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

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

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

- To dispatch the same event to two different functions.

  See the next section.

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

  When you pass a function as a parameter to `addEventListener()`, the function is invoked in the scope of the broadcaster component instance, not the object in which it is declared. See "Delegating the scope of a function" on page 76.

Delegating events to functions

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

To use the following code, place a check box named `myCheckBox_chb` and a button named `myButton_btn` on the Stage. Select both instances and press F8 to create a new symbol. Click Advanced if the dialog box is in basic mode, and select Export for ActionScript. Enter **Cart** in the AS 2.0 Class text box. In the Property inspector, set the instance name for the new symbol to anything you want. The symbol is now associated with the Cart class and an instance of the symbol becomes an instance of this class.

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

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

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

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

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

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

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

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

// two separate functions handle the events

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

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

Delegating the scope of a function

The `addEventListener()` method requires two parameters: the name of an event and a reference to a listener. The listener can either be an object or a function. If you pass an object, the callback function assigned to the object is invoked in the scope of the object. However, if you pass a function, the function is invoked in the scope of the component instance that calls `addEventListener()`. (For more information, see "About scope in listeners" on page 71.)

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

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

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

class Cart {

  var myCheckBox_chb:CheckBox;
  var myButton_btn:Button;

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

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

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

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

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

class Cart {
  var myCheckBox_chb:CheckBox;
  var myButton_btn:Button;
  // define a variable to access
  // from the chb_onClick function
  var i:Number = 10

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

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

About the event object

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

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

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

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

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

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

listener = new Object();

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

Using the on() event handler

You can assign the `on()` event handler to a component instance, just as you would assign a handler to a button or movie clip. An `on()` event handler can be useful for simple testing, but for all applications, use event listeners, instead. For more information, see "Using listeners to handle events" on page 64.

When you use the keyword `this` within an `on()` handler attached directly to a component (assigned to the component *instance* in the Actions panel), `this` refers to the component instance. For example, the following code, attached directly to the Button component instance `myButton`, displays "_level0.myButton" in the Output panel:

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

To use the on() event handler:

1. Drag a User Interface component to the Stage.

   For example, drag a Button component to the Sage.

2. On the Stage, select the component and open the Actions panel.

3. Add the `on()` handler to the Actions panel in the format:

   ```
   on(event){
     //your statements go here
   }
   ```

   For example:

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

   Flash runs the code inside the `on()` handler when the event for the `on()` handler occurs (in this case, a button click).

4. Select Control > Test Movie and click the button to see the output.

CHAPTER 5

Customizing Components

5

You might want to change the appearance of components as you use them in different applications. You can customize component appearance using the following three approaches, individually or in combination:

Styles User interface (UI) components have style properties that set the appearance of some aspects of a component. Each component has its own set of modifiable style properties, and not all visual aspects of a component can be changed by setting a style. For more information, see "Using styles to customize component color and text" on page 82.

Skins A *skin* comprises the collection of symbols that make up the graphical display of a component. *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. Some aspects of a component that cannot be set through its style properties can be set by modifying the skin. For more information, see "About skinning components" on page 96.

Themes A theme is a collection of both styles and skins that you can save as a FLA file and apply to another document. For more information, see "About themes" on page 108.

This chapter contains the following sections:

Using styles to customize component color and text 82
About skinning components 96
About themes 108
Combining skinning and styles to customize a component .118

Using styles to customize component color and text

Flash provides style properties that you can edit for every UI component. Within the documentation for each specific component, you'll see a table that lists the modifiable styles for that component (for example, you can see a table of styles for the Accordion component in "Using styles with the Accordion component" in the *Components Language Reference*). Additionally, UI components inherit the `setStyle()` and `getStyle()` methods from the UIObject class (see `UIObject.setStyle()` and `UIObject.getStyle()`). For a component instance, you can use the `setStyle()` and `getStyle()` methods to set and get style property values, as shown later in "Setting styles on a component instance" on page 84.

NOTE	You cannot set styles for the media components.

Using style declarations and themes

In a broader scope, styles are organized within style *declarations* where you can control style property values across multiple component instances. A style declaration is an object created by the CSSStyleDeclaration class, and its properties are the style settings you can assign to components. Style declarations in ActionScript are modeled after the way "cascading style sheets" (CSS) affect HTML pages. For HTML pages, you can create a style sheet file that defines style properties for the content in a group of HTML pages. With components, you can create a style declaration object and add style properties to that style declaration object to control the appearance of components in a Flash document.

Furthermore, style declarations are organized within *themes*. Flash provides two visual themes for components: Halo (HaloTheme.fla) and Sample (SampleTheme.fla). A *theme* is a set of styles and graphics that controls the appearance of components in a document. Each theme provides styles to the components. The styles used by each component depend in part on what theme the document uses. Some styles, such as `defaultIcon`, are used by the associated components regardless of the theme applied to the document. Other styles, such as `themeColor` and `symbolBackgroundColor`, are used only by components if the corresponding theme is in use. For example, `themeColor` is used only if the Halo theme is in use, and `symbolBackgroundColor` is used only if the Sample theme is in use. To determine what style properties you can set for a component, you must know which theme is assigned to that component. The style tables for each component in *Components Language Reference* indicate whether each style property applies to one or both of the supplied themes. (For more information, see "About themes" on page 108.)

Understanding style settings

As you use styles and style declarations, you'll notice that you can set styles in various ways (at the global, theme, class, style declaration, or style property levels). And, some style properties may be inherited from a parent component (for example, an Accordion child panel may inherit a font treatment from the Accordion component). Here are a few key points about style behavior:

Theme dependence The style properties you can set on a particular component are determined by the current theme. By default, Flash components are designed to use the Halo theme, but Flash also provides a Sample theme. So, when you read a style properties table, like the one for the Button component in "Using styles with the Button component" in the *Components Language Reference*, notice which theme supports the style you want. The table indicates Halo, Sample, or Both (meaning both themes support the style property). To change the current theme, see "Switching themes" on page 108.

Inheritance You cannot set inheritance within ActionScript. A component child is designed either to inherit a style from the parent component, or not.

Global style sheets Style declarations in Flash don't support "cascading" for Flash documents the way CSS does for HTML documents. All style sheet declaration objects are defined at the application (global) level.

Precedence If a component style is set in more than one way (for example, if `textColor` is set at the global level and at the component instance level), Flash uses the first style it encounters according to the order listed in "Using global, custom, and class styles in the same document" on page 92.

Setting styles

The existence of style properties, their organization within style declarations, and the broader organization of style declarations and graphics into themes enables you 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.

  For more information, see "Setting styles on a component instance" on page 84.

- Adjust 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.

  For more information, see "Setting global styles" on page 86.

- Create custom style declarations and apply them to several component instances.

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

  For more information, see "Setting custom styles for groups of components" on page 87.
- Create default class style declarations.

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

  For more information, see "Setting styles for a component class" on page 89.
- Use inheriting styles to set styles for components in a portion of a document.

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

  For more information, see "Setting inheriting styles on a container" on page 90.

Flash does not display changes made to style properties when you view components on the Stage using the Live Preview feature. For more information, see "Components in Live Preview" on page 60.

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

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

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

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

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

You can also access the styles directly as properties (for example, `myButton.color = 0xFF00FF`).

Style properties set on a component instance through `setStyle()` have the highest priority and override all other style settings based on style declaration or theme. However, the more properties you set using `setStyle()` on a single component instance, the slower the component will render at runtime. You can speed the rendering of a customized component with ActionScript that defines the style properties during the creation of the component instance using `UIObject.createClassObject()` in the *Components Language Reference*, and placing the style settings in the *`initObject`* parameter. For example, with a ComboBox component in the current document library, the following code creates a combo box instance named my_cb, and sets the text in the combo box to italic and aligned right:

```
createClassObject(mx.controls.ComboBox, "my_cb", 1, {fontStyle:"italic",
  textAlign:"right"});
my_cb.addItem({data:1, label:"One"});
```

> **NOTE**
> If you want to change multiple properties, or change properties for multiple component instances, you can create a custom style. A component instance that uses a custom style for multiple properties will render faster than a component instance with several `setStyle()` calls. For more information, see "Setting custom styles for groups of components" on page 87.

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

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

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

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

   For a list of styles supported by a particular component, see the component's entry in the *Components Language Reference*.

To create a component instance and set multiple properties simultaneously using ActionScript:

1. Drag a component to the library.
2. Open the Actions panel and select Scene 1, then select Layer 1: Frame 1.
3. Enter the following syntax to create an instance of the component and set its properties:

   ```
   createClassObject(className, "instance_name", depth, {style:"setting",
     style:"setting"});
   ```

   So, for example, with a Button component in the library, the following ActionScript creates a button instance `my_button` at depth `1` with the text styles set to purple and italicized:

   ```
   createClassObject(mx.controls.Button, "my_button", 1, {label:"Hello",
     color:"0x9900CC", fontStyle:"italic"});
   ```

   For more information, see `UIObject.createClassObject()`.
4. Select Control > Test Movie to view the changes.

   For a list of styles supported by a particular component, see the component's entry in the *Components Language Reference*.

Setting global styles

By default, all components adhere to a global style declaration until another style declaration is attached to the component (as in "Setting custom styles for groups of components" on page 87). The global style declaration is assigned to all Flash components built with version 2 of the Macromedia Component Architecture. The _global object has a `style` property (`_global.style`) that is an instance of CSSStyleDeclaration, and acts as the global style declaration. If you change a style property's value on the global style declaration, the change is applied to all components in your Flash document.

> **CAUTION**
>
> Some styles are set on a component class's CSSStyleDeclaration instance (for example, the `backgroundColor` style of the TextArea and TextInput components). Because the class style declaration takes precedence over the global style declaration when style values are determined, setting `backgroundColor` on the global style declaration would have no effect on the TextArea and TextInput components. For more information about style precedence, see "Using global, custom, and class styles in the same document" on page 92. For more information about editing a component class's CSSStyleDeclaration, see "Setting styles for a component class" on page 89.

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 50.

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

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

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

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

Setting custom styles for groups of components

You can create custom style declarations to specify a unique set of properties for groups of components in your Flash document. In addition to the _global object's `style` property (discussed in "Setting global styles" on page 86), which determines the default style declaration for an entire Flash document, the _global object also has a `styles` property, which is a list of available custom style declarations. So, you can create a style declaration as a new instance of the CSSStyleDeclaration object, assign it a custom style name, and place it in the `_global.styles` list. Then, you specify the properties and values for the style, and assign the style name to component instances that should share the same look.

Keep in mind that when you assign the style name to a component instance, the component responds only to style properties that component supports. For a list of the style properties each component supports, see the individual component entries in the *Components Language Reference*.

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

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

Custom style settings have priority over class, inherited, and global style settings. For a list of style precedence, see "Using global, custom, and class styles in the same document" on page 92.

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

1. Add at least one component to the Stage.

   For more information, see "Adding components to Flash documents" on page 50.

   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 8.

2. Select a frame in the Timeline on which (or before which) the component appears.
3. Open the Actions panel.
4. Add the following import statement so you will have access to the constructor function for creating a new style declaration from within the CSSStyleDeclaration class:

   ```
   import mx.styles.CSSStyleDeclaration;
   ```

5. Use the following syntax to create an instance of the CSSStyleDeclaration object to define the new custom style format:

   ```
   var new_style:Object = new CSSStyleDeclaration();
   ```

6. Give your style declaration a name, like "myStyle," in the `_global.styles` list of custom style declarations, and identify the object containing all the properties for your new style declaration.

   ```
   _global.styles.myStyle = new_style;
   ```

7. Use the `setStyle()` method (inherited from the UIObject class) to add properties to the `new_style` object, which are in turn associated with the custom style declaration `myStyle`:

   ```
   new_style.setStyle("fontFamily", "_serif");
   new_style.setStyle("fontSize", 14);
   new_style.setStyle("fontWeight", "bold");
   new_style.setStyle("textDecoration", "underline");
   new_style.setStyle("color", 0x666699);
   ```

8. In the same Script pane, use the following syntax to set the `styleName` property of three specific components to the custom style declaration name:

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

You *can* also access styles on the custom style declaration using the `setStyle()` and `getStyle()` methods through the declaration's global `styleName` property. For example, the following code sets the `backgroundColor` style on the `myStyle` style declaration:

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

However, steps 5 and 6 associated the `new_style` instance with the style declaration so you can use the shorter syntax, like `new_style.setStyle("themeColor", "haloOrange")`.

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

Setting styles for a component class

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

CAUTION

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

The following code first checks to see if the current theme already has a style declaration for CheckBox, and, if not, creates a new one. Then the code uses the `setStyle()` method to define a style property for the CheckBox style declaration (in this case, "color" sets the color for all check box label text to blue):

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

For a table of the style properties you can set on the CheckBox component, see "Using styles with the CheckBox component" in the *Components Language Reference*.

Custom style settings have priority over inherited and global style settings. For a list of style precedence, see "Using global, custom, and class styles in the same document" on page 92.

Setting inheriting styles on a container

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

The following styles are inheriting styles:

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

The Style Manager tells Flash whether a style inherits its value. Additional styles can also be added at runtime as inheriting styles. For more information, see StyleManager class in the *Components Language Reference*.

> **NOTE** One major difference between the implementation of styles for Flash components, and cascading style sheets for HTML pages, is that the CSS `inherit` value is not supported for Flash components. Styles are either inherited or not by component design.

Inherited styles take priority over global styles. For a list of style precedence, see "Using global, custom, and class styles in the same document" on page 92.

The following example demonstrates how inheriting styles can be used with an Accordion component, which is available with Flash Professional 8. (The inheriting styles feature is supported by both Flash Basic 8 and Flash Professional 8.)

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

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

   By dragging the components to the library, you make them available to your script at runtime.
5. Add the following ActionScript to the first frame of the Timeline:

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

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

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

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

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

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

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

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

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

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

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

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

1. Flash looks for a style property on the component instance.
2. Flash looks at the `styleName` property of the instance to see if a custom style declaration is assigned to it.
3. Flash looks for the property on a default class style declaration.
4. If the style is one of the inheriting styles, Flash looks through the parent hierarchy for an inherited value.
5. Flash looks for the style in the global style declaration.
6. If the property is still not defined, the property has the value `undefined`.

About color style properties

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

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

Color names are strings that map to commonly used colors. You can add new color names by using the Style Manager (see StyleManager class in the *Components Language Reference*). The following table lists the default color names:

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

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

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

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

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

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

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

Customizing component animations

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

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

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

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

Easing class	Description
Back	Extends beyond the transition range at one or both ends one time to provide a slight overflow effect.
Bounce	Provides a bouncing effect entirely within the transition range at one or both ends. The number of bounces is related to the duration: longer durations produce more bounces.
Elastic	Provides an elastic effect that falls outside the transition range at one or both ends. The amount of elasticity is unaffected by the duration.
None	Provides an equal movement from start to end with no effects, slowing, or speeding. This transition is also commonly referred to as a *linear transition*.

Easing class	Description
Regular	Provides for slower movement at one or both ends for a speeding-up effect, a slowing-down effect, or both.
Strong	Provides for much slower movement at one or both ends. This effect is similar to Regular but is more pronounced.

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

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

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

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

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

For more information see "Applying easing methods to components" in the *Components Language Reference.*

Getting style property values

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

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

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

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

About skinning components

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

Version 2 components are compiled clips—you cannot see their assets in the library. However, the Flash installation includes FLA files that contain all the component skins. These FLA files are called *themes*. Each theme has a different appearance and behavior, but contains skins with the same symbol names and linkage identifiers. This lets you drag a theme onto the Stage in a document to change its appearance. You also use the theme FLA files to edit component skins. The skins are located in the Themes folder in the Library panel of each theme FLA file. (For more information about themes, see "About themes" on page 108.)

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

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

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

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

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

- To change the skins associated with all instances of a particular component in a single document, copy and modify individual skin elements. (See "Editing component skins in a document" on page 97).

  This method of skinning is recommended for beginners, because it doesn't require any scripting.
- To replace all the skins in a document with a new set (with each kind of component sharing the same appearance), apply a theme. (See "About themes" on page 108.)

  This method of skinning is recommended for applying a consistent look and feel across all components and across several documents.
- To link the color of a skin element to a style property, add ActionScript code to the skin to register it as a colored skin element. (See "Linking skin color to styles" on page 100).
- To use different skins for multiple instances of the same component, create new skins and set skin properties. (See "Creating new component skins" on page 99 and "Applying new skins to a component" on page 101.)
- To change skins in a subcomponent (such as a scroll bar in a List component), subclass the component. (See "Applying new skins to a subcomponent" on page 103.)
- 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 a subcomponent" on page 106.)

Editing component skins in a document

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

The procedure described below is very similar to creating and applying a new theme (see "About themes" on page 108). The primary difference is that this procedure describes copying symbols directly from the theme already in use to a single document and editing only a small number of all skins available. This is appropriate when your modifications are all in a single document and when you are modifying skins for only a few components. If the edited skins will be shared in multiple documents or encompass changes in several components, you may find editing the skins easier if you create a new theme.

An article on advanced skinning can be found online in the Macromedia Developer Center at www.macromedia.com/devnet/mx/flash/articles/skinning_2004.html.

To edit component skins in a document:

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

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 108.
3. In the theme's Library panel, select Flash UI Components 2/Themes/MMDefault and drag the Assets folder of any components in your document to the library for your document.

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

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

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

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

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

   > **NOTE** The live preview of the components on the Stage will not reflect the edited skins.

9. Select Control > Test Movie.

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

Creating new component skins

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

To create a new skin:

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

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

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

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

   For more information, see "Applying new skins to a component" on page 101, "Applying new skins to a subcomponent" on page 103, or "Changing skin properties in a subcomponent" on page 106.

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

Linking skin color to styles

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

To link a skin to a style property:

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

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 108.
3. In the theme's Library panel, select Flash UI Components 2/Themes/MMDefault, and drag the Assets folder of any components in your document to the library for your document.

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

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

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

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

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

   For this example, name the instance **innerCircle**.

8. Add ActionScript code to register the skin element or a movie clip instance it contains as a colored skin element.

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

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

   In this example you're using a color that already corresponds to an existing style name in the Sample style. Wherever possible, it's best to use style names corresponding to official Cascading Style Sheet standards or styles provided by the Halo and Sample themes.

9. Repeat steps 5-8 until you've edited all the skins you want to change.

   For this example, repeat these steps for the RadioTrueDisabled skin, but instead of converting the existing graphic to a movie clip, delete the graphic and drag the existing Inner Circle symbol to the RadioTrueDisabled skin element.

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

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

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

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

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

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

13. Select Control > Test Movie.

Applying new skins to a component

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

To dynamically create a component and apply a new skin:

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

   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 library of DynamicSkinning.fla.

   This adds the symbols to the document's library, but doesn't make them visible in the document.
5. Open the Actions panel and enter the following on Frame 1:

   ```
   import mx.controls.RadioButton;
   createClassObject(RadioButton, "myRadio", 0,
     {trueUpIcon:"MyRadioTrueUp", label: "My Radio Button"});
   ```
6. Select Control > Test Movie.

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

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

   This adds the symbols to the document's library, but doesn't make them visible in the document.
5. Select the RadioButton component on the Stage and open the Actions panel.
6. Attach the following code to the RadioButton instance:

   ```
   onClipEvent(initialize){
     trueUpIcon = "MyRadioTrueUp";
   }
   ```
7. Select Control > Test Movie.

Applying new skins to a subcomponent

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

If a component is composed of subcomponents, the subcomponents are identified in the component's entry in the *Components Language Reference*.

To apply a new skin to a subcomponent:

1. Follow the steps in "Creating new component skins" on page 99, but edit a scroll bar skin. For this example, edit the ScrollDownArrowDown skin and give it the new name **MyScrollDownArrowDown**.
2. Select File > New to create a new Flash document.
3. Select File > Save and give the file a unique name, such as **SubcomponentProject.fla**.
4. Drag the List component in the Components panel to the library.

   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 library of SubcomponentProject.fla.

   This adds the symbol to the Library panel, but doesn't make it visible in the document.
6. Do one of the following:
   - If you want to change all scroll bars in a document, enter the following code in the Actions panel on Frame 1 of the Timeline:

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

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

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

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

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

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

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

7. Select Control > Test Movie.

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

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

1. Follow the steps in "Creating new component skins" on page 99, 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 library of SkinsInitExample.fla.

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

   Export in First Frame should be automatically selected; if it is not, select it.
6. Double-click MyScrollDownArrowDown in the library to open it in symbol-editing mode.
7. Enter the following code on Frame 1 of the MyScrollDownArrowDown symbol:

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

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

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

     > **NOTE** Add enough data so that the vertical scroll bar appears, or set `vScrollPolicy` to `true`.

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

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

1. Follow the steps in "Editing component skins in a document" on page 97, but edit a scroll bar skin. For this example, edit the ScrollDownArrowDown skin and give it the new name **MyScrollDownArrowDown**.
2. Select File > New and create a Flash document.
3. Select File > Save and give the file a unique name, such as **MyVScrollTest.fla.**
4. Drag MyScrollDownArrowDown from the theme library to the MyVScrollTest.fla library.
5. Select Insert > New Symbol and give the symbol a unique name, such as **MyVScrollBar**.
6. Select the Export for ActionScript check box. Click OK.

   Export in First Frame should be automatically selected; if it is not, select it.
7. Enter the following code on Frame 1 of the `MyVScrollBar` symbol:

   ```
   #initclip 10
     import MyVScrollBar
     Object.registerClass("VScrollBar", MyVScrollBar);
   #endinitclip
   ```
8. Drag a List component from the Components panel to the Stage.
9. In the Property inspector, enter as many Label parameters as necessary for the vertical scroll bar to appear.
10. Select File > Save.
11. Select File > New and create a new ActionScript file.

12. Enter the following code:

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

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

Changing skin properties in a subcomponent

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

If a component is composed of subcomponents, the subcomponents are identified in the component's entry in the *Components Language Reference*.

To skin a subcomponent:

1. Follow the steps in "Editing component skins in a document" on page 97, 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 library of MyComboTest.fla.

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

6. Select the Export for ActionScript check box and click OK.

   Export in First Frame should be automatically selected; if it is not, select it.

7. Enter the following code in the Actions panel on Frame 1 of the `MyComboBox` symbol:

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

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

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

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

About themes

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

- In Windows: C:\Program Files\Macromedia\Flash 8*language*\Configuration\ComponentFLA\
- On the Macintosh: HD/Applications/Macromedia Flash 8/Configuration/ComponentFLA/

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

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

A theme can also contain a new set of style default values. You must write ActionScript code to create a global style declaration and any additional style declarations. For more information, see "Modifying default style property values in a theme" on page 112.

Switching themes

Macromedia Flash installs two themes: Halo and Sample. You'll notice that the component reference information for each component contains a table of style properties you can set for either (or both) themes. So, when you read a style properties table, like the one for the Button component in "Using styles with the Button component" in the *Components Language Reference*, notice which theme supports the style you want. The table indicates Halo, Sample, or Both (meaning both themes support the style property).

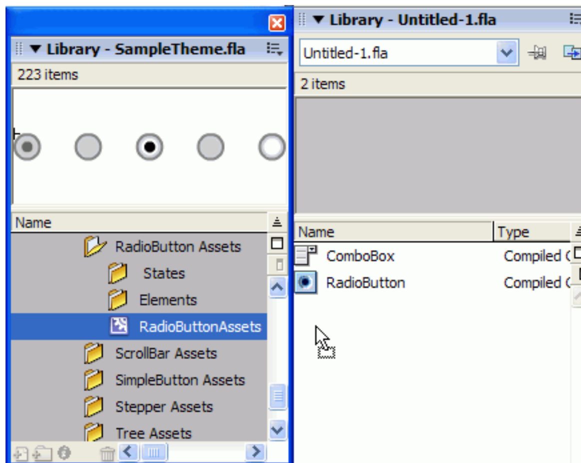
The Halo theme is the default theme for components. So, if you want to use the Sample theme, you need to switch the current theme from Halo to Sample.

To switch to the Sample theme:

1. Select File > Open and open the document that uses version 2 components in Flash, or select File > New and create a new document that uses version 2 components.
2. Select File > Import > Open External Library, and select SampleTheme.fla to apply to your document.

   This file is located in the application-level configuration folder. For the exact location on your operating system, see "About themes" on page 108.
3. In the SampleTheme.fla theme's Library panel, select Flash UI Components 2/Themes/MMDefault and drag the Assets folders of any components in your document to the Library panel of your Flash document.

   For example, drag the RadioButton Assets folder to your library.



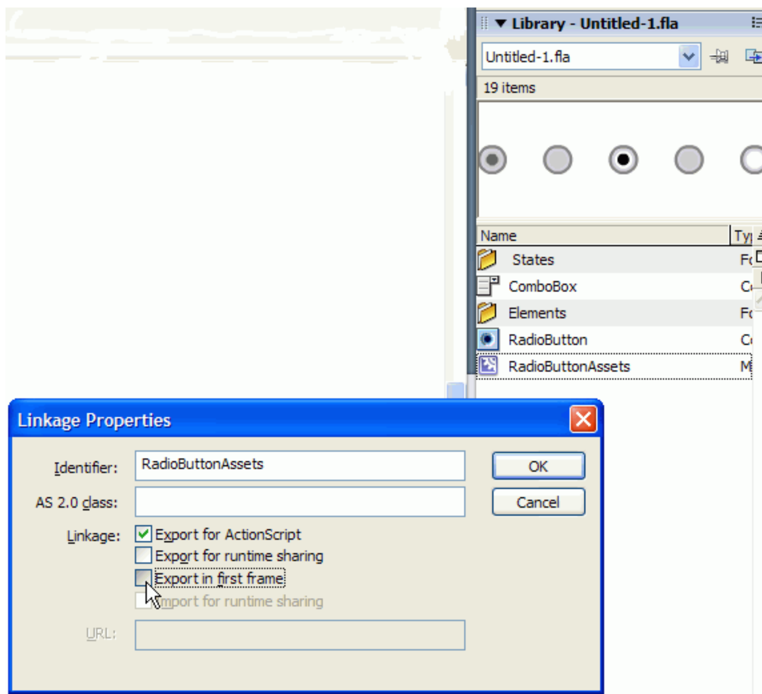
If you're unsure about which components are in the document, drag the entire Sample Theme movie clip to the Stage. The skins are automatically assigned to components in the document.

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

4. If you drag individual component Assets folders to the Library panel of your document, make sure the Assets symbol for each component is set to Export in First Frame.

   For example, the Assets folder for the RadioButton component is called RadioButton Assets. Open the RadioButtonAssets folder, and you'll see a movie clip symbol called RadioButtonAssets. The RadioButtonAssets symbol contains all of the individual asset symbols within it.

   Right-click (Windows) or Control-click (Macintosh) the RadioButtonAssets symbol in the library of your document, and select the Linkage menu option. Check Export in First Frame, so all individual asset symbols will also export in the first frame. Then, click OK to save the settings.



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.

For a list of the skins supported by each component and their properties, see the *Components Language Reference*.

To create a new theme:

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

   Give the copy a unique name such as **MyTheme.fla**.
2. Select File > Open MyTheme.fla in Flash.
3. Select Window > Library to open the library if it isn't open already.
4. Double-click any skin symbol you want to modify to open it in symbol-editing mode.

   The skins are located in the Flash UI Components 2/Themes/MMDefault/*Component* Assets folder (this example uses RadioButton Assets).
5. Modify the symbol or delete the graphics and create new graphics.

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

   For example, open the States/RadioFalseDisabled asset and change the inner circle to a light gray.
6. When you finish editing the skin symbol, click the Back button at the left side of the information bar at the top of the Stage to return to document-editing mode.
7. Repeat steps 4-6 until you've edited all the skins you want to change.
8. Apply MyTheme.fla to a document by following the steps shown later in this chapter. (See "Applying a new theme to a document" on page 113.)

Modifying default style property values in a theme

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

To modify default style values in a theme:

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

   For example, create a folder called **myTheme**.
2. Copy an existing Defaults class to your new theme folder.

   For example, copy mx/skins/halo/Defaults.as to mx/skins/myTheme/Defaults.as.
3. Open the new Defaults class in an ActionScript editor.

   Flash Professional 8 users can open the file within Flash. Or, you can open the file in Notepad in Windows or SimpleText on the Macintosh.
4. Modify the class declaration to reflect the new package.

   For example, our new class declaration is `class mx.skins.myTheme.Defaults`.
5. Modify the style settings as desired.

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

   ```
   o.disabledColor = 0x663333;
   ```
6. Save the changed Defaults class file.
7. Copy an existing FocusRect class from the source theme to your custom theme.

   For example, copy mx/skins/halo/FocusRect.as to mx/skins/myTheme/FocusRect.as.
8. Open the new FocusRect class in an ActionScript editor.
9. Modify all references to the source theme's package to the new theme's package.

   For example, change all occurrences of "halo" to "myTheme."
10. Save the changed FocusRect class file.
11. Open the FLA file for your custom theme.

    This example uses MyTheme.fla.
12. Open the library (Window > Library) and locate the Defaults symbol.

    In this example, it's in Flash UI Components 2/Themes/MMDefault/Defaults.
13. Edit the symbol properties for the Default symbol.
14. Change the AS 2.0 Class setting to reflect your new package.

    The example class is mx.skins.myTheme.Defaults.
15. Click OK.

16. Locate the FocusRect symbol.

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

17. Edit the symbol properties for the FocusRect symbol.

18. Change the AS 2.0 Class setting to reflect your new package.

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

19. Click OK.

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

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

In this example you used a new theme to customize the text color of disabled components. This particular customization, changing a single default style property value, would have been accomplished more easily through styling as explained in "Using styles to customize component color and text" on page 82. Using a new theme to customize defaults is appropriate when customizing many style properties or when already creating a new theme to customize component graphics.

Applying a new theme to a document

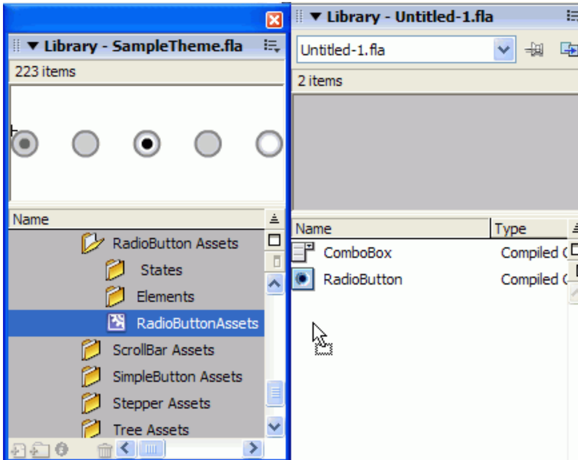
To apply a new theme to a document, open a theme FLA file as an external library, and drag the theme folder from the external library to the document library. The following steps explain the process in detail, assuming you already have a new theme (for more information, see "Creating a new theme" on page 111).

To apply a theme to a document:

1. Select File > Open and open the document that uses version 2 components in Flash, or select File > New and create a new document that uses version 2 components.
2. Select File > Import > Open External Library, and select the FLA file of the theme you want to apply to your document.

3. In the theme's Library panel, select Flash UI Components 2/Themes/MMDefault and drag the Assets folders for any components you want to use to your document's library.

   For example, drag the RadioButton Assets folder to your library.

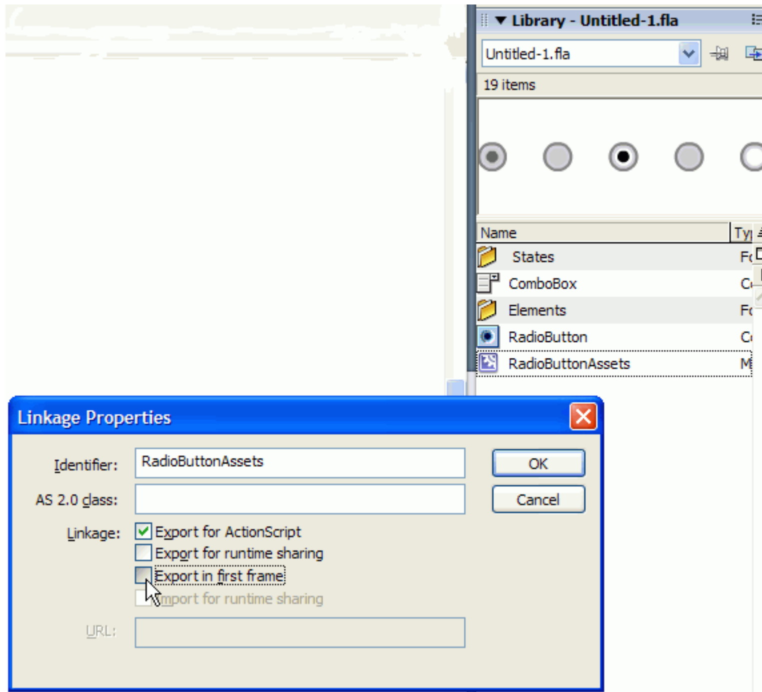


   If you're unsure which components are in the document, drag the entire theme movie clip (for example, for the SampleTheme.fla, the main theme movie clip is Flash UI Components 2 > SampleTheme) to the Stage. The skins are automatically assigned to components in the document.

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

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

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



5. Select Control > Test Movie to see the new theme applied.

Changing export settings

When you apply the Sample or Halo theme to your document, many of the skin assets are set to export in the first frame in order to make them immediately available to the components during playback. However, if you change the publishing export setting (File > Publish Settings > Flash tab > ActionScript Version Settings button > Export Frame for Classes) of your FLA file to a frame after the first frame, you must also change the export settings for the assets in the Sample and Halo themes. To do this, you must open the following component assets in your document's library and deselect the Export in First Frame check box (right-click > Linkage > Export in First Frame):

Sample theme

- Flash UI Components 2/Base Classes/UIObject
- Flash UI Components 2/Themes/MMDefault/Defaults
- Flash UI Components 2/Base Classes/UIObjectExtensions
- Flash UI Components 2/Border Classes/BoundingBox
- Flash UI Components 2/SampleTheme
- Flash UI Components 2/Themes/MMDefault/Button Assets/Elements/ButtonIcon
- Flash UI Components 2/Themes/MMDefault/DateChooser Assets/ Elements/ Arrows/cal_disabledArrow
- Flash UI Components 2/Themes/MMDefault/FocusRect
- Flash UI Components 2/Themes/MMDefault/Window Assets/ States/ CloseButtonOver
- Flash UI Components 2/Themes/MMDefault/Accordion Assets/ AccordionHeaderSkin
- Flash UI Components 2/Themes/MMDefault/Alert Assets/AlertAssets
- Flash UI Components 2/Themes/MMDefault/Border Classes/Border
- Flash UI Components 2/Themes/MMDefault/Border Classes/CustomBorder
- Flash UI Components 2/Themes/MMDefault/Border Classes/RectBorder
- Flash UI Components 2/Themes/MMDefault/Button Assets/ActivatorSkin
- Flash UI Components 2/Themes/MMDefault/Button Assets/ButtonSkin

Halo theme

- Flash UI Components 2/Base Classes/UIObject
- Flash UI Components 2/Themes/MMDefault/Defaults
- Flash UI Components 2/Base Classes/UIObjectExtensions
- Flash UI Components 2/Component Assets/BoundingBox
- Flash UI Components 2/HaloTheme
- Flash UI Components 2/Themes/MMDefault/Accordion Assets/AccordionHeaderSkin
- Flash UI Components 2/Themes/MMDefault/Alert Assets/AlertAssets
- Flash UI Components 2/Themes/MMDefault/Border Classes/Border
- Flash UI Components 2/Themes/MMDefault/Border Classes/CustomBorder
- Flash UI Components 2/Themes/MMDefault/Border Classes/RectBorder
- Flash UI Components 2/Themes/MMDefault/Button Assets/ActivatorSkin
- Flash UI Components 2/Themes/MMDefault/Button Assets/ButtonSkin
- Flash UI Components 2/Themes/MMDefault/Button Assets/Elements/ButtonIcon
- Flash UI Components 2/Themes/MMDefault/CheckBox Assets/Elements/CheckThemeColor1
- Flash UI Components 2/Themes/MMDefault/CheckBox Assets/CheckBoxAssets
- Flash UI Components 2/Themes/MMDefault/ComboBox Assets/ComboBoxAssets
- Flash UI Components 2/Themes/MMDefault/DataGrid Assets/DataGridAssets
- Flash UI Components 2/Themes/MMDefault/DateChooser Assets/DateChooserAssets
- Flash UI Components 2/Themes/MMDefault/FocusRect
- Flash UI Components 2/Themes/MMDefault/Menu Assets/MenuAssets
- Flash UI Components 2/Themes/MMDefault/MenuBar Assets/MenuBarAssets
- Flash UI Components 2/Themes/MMDefault/ProgressBar Assets/ProgressBarAssets
- Flash UI Components 2/Themes/MMDefault/RadioButton Assets/Elements/RadioThemeColor1
- Flash UI Components 2/Themes/MMDefault/RadioButton Assets/Elements/RadioThemeColor2
- Flash UI Components 2/Themes/MMDefault/RadioButton Assets/RadioButtonAssets
- Flash UI Components 2/Themes/MMDefault/ScrollBar Assets/HScrollBarAssets
- Flash UI Components 2/Themes/MMDefault/ScrollBar Assets/ScrollBarAssets

- Flash UI Components 2/Themes/MMDefault/ScrollBar Assets/VScrollBarAssets
- Flash UI Components 2/Themes/MMDefault/Stepper Assets/Elements/StepThemeColor1
- Flash UI Components 2/Themes/MMDefault/Stepper Assets/NumericStepperAssets
- Flash UI Components 2/Themes/MMDefault/Tree Assets/TreeAssets
- Flash UI Components 2/Themes/MMDefault/Window Assets/Window Assets

Combining skinning and styles to customize a component

In this section, you will customize a combo box component instance using styles, themes, and skinning settings. The procedures demonstrate how to combine skinning with style settings to create a unique presentation for a component.

Creating a component instance on the Stage

The first part of this exercise requires you to create a ComboBox instance for customizing.

To create the ComboBox instance:

1. Drag a ComboBox component to the Stage.
2. In the Properties panel, name the instance **my_cb**.
3. In the first frame of the main Timeline, add the following ActionScript (make sure you are adding it to the frame and not the component, itself; the Actions panel should say "Actions - Frame" in the title bar):

```
my_cb.addItem({data:1, label:"One"});
my_cb.addItem({data:2, label:"Two"});
```

4. Select Control > Test Movie to see the combo box with the default style and skinning from the Halo theme.

Creating the new style declaration

Now, you need to create a new style declaration and assign styles to the style declaration. After you have all the styles you want in the style declaration, you can assign the new style name to the combo box instance.

To create a new style declaration and give it a name:

1. In the first frame of the main Timeline, add the following line at the beginning of your ActionScript (as a coding convention, you should place all import statements at the beginning of your ActionScript):

   ```
   import mx.styles.CSSStyleDeclaration;
   ```

2. On the next line, name the new style declaration and add it to the global style definitions:

   ```
   var new_style:Object = new CSSStyleDeclaration();
   _global.styles.myStyle = new_style;
   ```

   After you assign a new style declaration to the _global style sheet, you can attach individual style settings to the `new_style` style declaration. For more information about creating a style sheet for groups for components, instead of style definitions for a single instance, see "Setting custom styles for groups of components" on page 87).

3. Attach some style settings to the `new_style` style declaration. The following style settings include style definitions available to the ComboBox component (see "Using styles with the ComboBox component" in the *Components Language Reference* for more a complete list) as well as styles from the RectBorder class, since the ComboBox component uses the RectBorder class:

   ```
   new_style.setStyle("textAlign", "right");
   new_style.setStyle("selectionColor", "white");
   new_style.setStyle("useRollOver", false);
   // borderStyle from RectBorder class
   new_style.setStyle("borderStyle", "none");
   ```

Assigning style definitions to the combo box

At this point, you have a style declaration containing a variety of styles, but you need to explicitly assign the style name to the component instance. You can assign this new style declaration to *any* component instance within your document in the following manner. Add the following line after the `addItem()` statements for `my_cb` (as a coding convention, you should keep all your combo box construction statements together):

```
my_cb.setStyle("styleName", "myStyle");
```

The ActionScript code attached to the first frame of the main Timeline should be as follows:

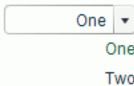
```
import mx.styles.CSSStyleDeclaration;

var new_style:Object = new CSSStyleDeclaration();
_global.styles.myStyle = new_style;

new_style.setStyle("textAlign", "right");
new_style.setStyle("selectionColor", "white");
new_style.setStyle("useRollOver", false);
// borderStyle from RectBorder class
new_style.setStyle("borderStyle", "none");

my_cb.addItem({data:1, label:"One"});
my_cb.addItem({data:2, label:"Two"});
my_cb.setStyle("styleName", "myStyle");
```

Select Control > Test Movie to see the styled combo box:



Changing the combo box theme

Every user interface component lists the style properties you can set for that component (for example, all the style properties you can set for a ComboBox component are listed in "Customizing the ComboBox component" in the *Components Language Reference*). Within the table of style properties, a column titled "Theme" shows which installed theme supports each style property. Not all the style properties are supported by all the installed themes. The default theme for all user interface components is the Halo theme. When you change the theme to the Sample theme, you can use a different set of style properties (some properties may no longer be available if they are listed as Halo only).

To change the theme for the styled component:

1. Select File > Import > Open External Library, and select SampleTheme.fla to open the Sample theme library in Flash.

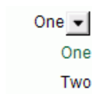
   This file is located in the application-level configuration folder:

   - In Windows: C:\Program Files\Macromedia\Flash 8*language*\Configuration\ComponentFLA\
   - On the Macintosh: HD/Applications/Macromedia Flash 8/Configuration/ComponentFLA/

2. Drag the main SampleTheme (Flash UI Components 2 > SampleTheme) movie clip from the SampleTheme library to your document's library.

   The ComboBox component is a combination of several components and classes, and requires assets from those other components and assets, including the Border and ScrollBar assets. The simplest way to ensure that you have all the assets from a theme that you need is to drag all the theme's assets to your library.

3. Select Control > Test Movie to see the styled combo box:



Editing the combo box skin assets

To edit the appearance of a component, edit the skins that comprise the component, graphically. To edit the skins, you open the component's graphic assets from within the current theme, and edit the symbols for that component. Macromedia recommends this approach because it doesn't remove or add symbols that other components might need; this approach edits the appearance of an existing component skin symbol.

> **NOTE**
> It is *possible*, but not recommended, to edit the source class files for a component so it uses symbols with different names as skins, and you can programmatically alter the ActionScript within a skin symbol (for an example of customized ActionScript and skin symbols, see "Customizing the Accordion component (Flash Professional only)" in the *Components Language Reference*). However, because several components, including the ComboBox component, share assets with other components, editing the source files or changing the skin symbol names can have unexpected results.

When you edit a component skin symbol:

- All instances of that component will use new skins (but not custom styles unless you explicitly attach the styles to the instances), *and* some components dependent on that component will use the new skins.
- If you assign a new theme after you've edited your component skins, make sure you don't overwrite the existing "edited" skins (a dialog box asks if you want to overwrite the skins and gives you an opportunity to stop Flash from overwriting the skins).

In this section, you will continue to use the combo box from the previous section (see "Changing the combo box theme" on page 120). The following steps change the appearance of the down arrow that opens the combo box menu from an arrow to a circle.

To edit the combo box down arrow symbol:

1. In your document's library, open the ComboBox assets to see the movie clips that are the skins for the button that opens and closes the combo box instance at runtime. Specifically, open the Themes > MMDefault > ComboBox Assets > States folder in your document's library.

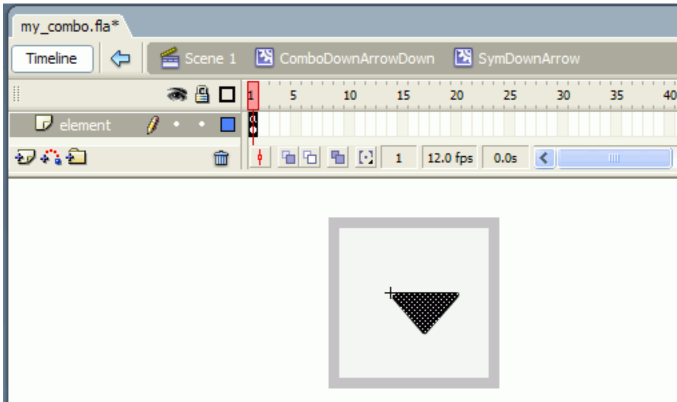
   The States folder contains four movie clips: ComboDownArrowDisabled, ComboDownArrowDown, ComboDownArrowOver, and ComboDownArrowUp. All four of these symbols are made up of other symbols. And all four use the same symbol for the down arrow (triangle), called SymDownArrow.

2. Double-click the ComboDownArrowDown symbol to edit it.

   You may need to zoom in, up to 800%, to see the details for the button.

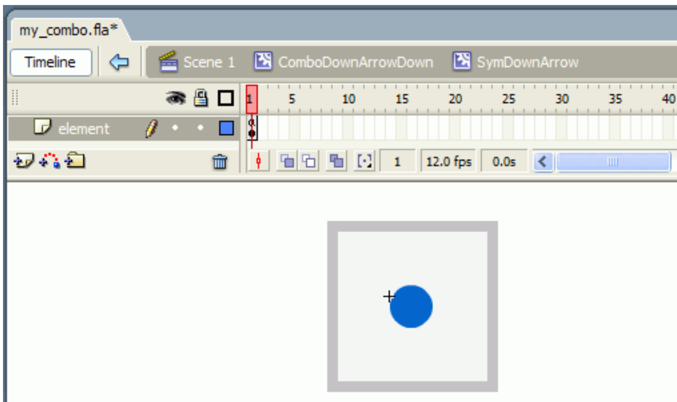
3. Double-click the down arrow (black triangle) to edit it.

NOTE	Make sure you have the symbol SymDownArrow selected, so you are deleting only the shape inside the movie clip and not the movie clip symbol, itself.

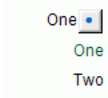


4. Delete the selected down arrow (the black triangle shape, *not the whole movie clip*) on the Stage.
5. While still editing SymDownArrow, draw a circle in the place where the down arrow had been.

   To make the change more noticeable, consider drawing a circle that is a bright color, like blue, approximately 4 pixels x 4 pixels, and with an *x* coordinate of `0` and a *y* coordinate of `-1` so it is centered.



6. Select Control > Test Movie to see the skinned combo box:



In your document's library, if you select ComboDownArrowOver and ComboDownArrowUp, you'll see that they also have the blue circle instead of the black triangle, because they also use SymDownArrow for the down arrow symbol.

CHAPTER 6

Creating Components

6

This chapter describes how to create your own component and package it for distribution.

This chapter contains the following sections:

Component source files 125
Overview of component structure. 126
Building your first component 127
Selecting a parent class 136
Creating a component movie clip 138
Creating the ActionScript class file 143
Incorporating existing components within your component 173
Exporting and distributing a component 182
Final steps in component development. 185

Component source files

The components available in the Components panel are precompiled SWC clips. The source Flash Document (FLA) containing the graphics and the source ActionScript class files (AS) containing the code for these components have also been provided for you to use in creating your own custom components. The source files for the version 2 components are installed with Macromedia Flash. It is helpful to open and review a few of these files, and try to understand their structure before you build your own components. The RadioButton component is a good example of a simpler component that you may want to explore first. All the components are symbols in the library of StandardComponents.fla. Each symbol is linked to an ActionScript class. Their location is as follows:

- FLA file source code
  - In Windows: C:\Program Files\Macromedia\
    Flash 8*language*\Configuration\ComponentFLA\StandardComponents.fla.
  - On the Macintosh: HD/Applications/Macromedia Flash 8/Configuration/
    ComponentFLA/StandardComponents.fla

- ActionScript class files
  - In Windows: C:\Program Files\Macromedia\Flash 8*language*\First Run\Classes\mx
  - On the Macintosh: HD/Applications/Macromedia Flash 8/First Run/Classes/mx

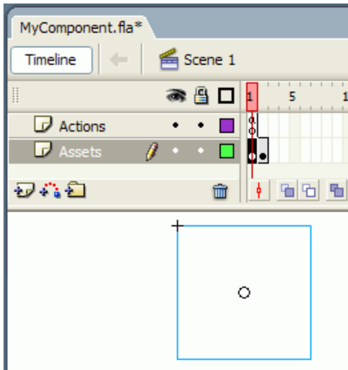
Overview of component structure

A component consists of a Flash (FLA) file and an ActionScript (AS) file. You can optionally create and package other files (for example, an icon and a .swd debugging file) with your component, but all components require a FLA and an ActionScript file. When you've finished developing your component, you export it as a SWC file.

A Flash (FLA) file, an ActionScript (AS) file, and a SWC file

The FLA file contains a movie clip symbol that must be linked to the AS file in both the Linkage Properties and the Component Definition dialog boxes.

The movie clip symbol has two frames and two layers. The first layer is an Actions layer and has a `stop()` global function on Frame 1. The second layer is an Assets layer with two keyframes: Frame 1 contains a bounding box; Frame 2 contains all other assets, including graphics and base classes, used by the component.



The ActionScript code specifying the properties and methods for the component is in a separate ActionScript class file. This class file also declares which, if any, classes the component extends. The name of the AS class file is the name of the component plus the ".as" extension. For example, MyComponent.as contains the source code for the MyComponent component.

It's a good idea to save the component's FLA and AS files in the same folder and give them the same name. If the AS file is not saved in the same folder, you must verify that the folder is in the classpath so the FLA file can find it. For more information about the classpath, see "Classes" in *Learning ActionScript 2.0 in Flash*.

Building your first component

In this section, you will build a Dial component. The completed component files, Dial.fla, Dial.as, and DialAssets.fla are located in the examples folder on your computer:

- In Windows: the C:\Program Files\Macromedia\Flash 8\Samples and Tutorials\Samples\Components\DialComponent folder.
- On the Macintosh: HD/Applications/Macromedia Flash 8/Samples and Tutorials/Samples/Components/DialComponent folder.

The Dial component is a potentiometer, like those used to measure potential difference in voltage. A user can click on the needle and drag it to change its position. The API for the Dial component has one property, `value`, that you can use to get and set the position of the needle.

This section takes you through the steps of creating a component. These procedures are discussed in more detail in subsequent sections (including "Selecting a parent class" on page 136, "Creating a component movie clip" on page 138, "Creating the ActionScript class file" on page 143, and "Exporting and distributing a component" on page 182).This section contains the following topics:

- "Creating the Dial Flash (FLA) file" on page 127
- "Creating the Dial class file" on page 130
- "Testing and exporting the Dial component" on page 133

Creating the Dial Flash (FLA) file

The first steps for creating a component include creating the component movie clip within a FLA document file.

To create the Dial FLA file:

1. In Flash, select File > New and create a new document.
2. Select File > Save As and save the file as **Dial.fla**.

   The file can have any name, but giving it the same name as the component is practical.
3. Select Insert > New Symbol. The component itself is created as a new MovieClip symbol so it will be available through the library.

   Name the component Dial, and assign it the behavior Movie clip.

4. If the Linkage section of the Create New Symbol dialog box isn't open, click the Advanced button to reveal it.
5. In the Linkage area, select Export for ActionScript and deselect Export in First Frame.
6. In the Identifier text box, enter a linkage identifier such as **Dial_ID**.
7. in the AS 2.0 Class text box, enter **Dial**. This value is the component class name. If the class is in a package (for example, mx.controls.Button), enter the entire package name.
8. Click OK.

   Flash changes to symbol-editing mode.
9. Insert a new layer. Name the top layer **Actions** and the bottom layer **Assets**.
10. Select Frame 2 in the Assets layer and insert a keyframe (F6).

    This is the structure of the component movie clip: an Actions layer and an Assets layer. The Actions layer has one keyframe and the Assets layer has two keyframes.
11. Select Frame 1 in the Actions layer and open the Actions panel (F9). Enter a `stop();` global function.

    This prevents the movie clip from proceeding to Frame 2.
12. Select File > Import > Open External Library and select the StandardComponents.fla file from the Configuration/ComponentFLA folder. For example:
    - In Windows: C:\Program Files\Macromedia\Flash 8*language*\Configuration\ComponentFLA\StandardComponents.fla.
    - On the Macintosh: HD/Applications/Macromedia Flash 8/Configuration/ComponentFLA/StandardComponents.fla

    NOTE: For information about folder locations, see "Configuration folders installed with Flash" in *Getting Started with Flash*.
13. Dial extends the UIComponent base class; therefore, you must drag an instance of the UIComponent into the Dial document. In the StandardComponents.fla library, browse to the UIComponent movie clip in the following folder: Flash UI Components 2 > Base Classes > FUIObject Subclasses and drag it to the Dial.fla library.

    Asset dependencies are automatically copied to the Dial library with UIComponent.

NOTE: When you drag UIComponent to the Dial library, the folder hierarchy in the Dial library is changed. If you plan to use your library again, or use it with other groups of components (such as the version 2 components), you should restructure the folder hierarchy to match the StandardComponents.fla library so that it's organized well and you avoid duplicate symbols.

14. In the Assets layer, select Frame 2 and drag an instance of UIComponent to the Stage.
15. Close the StandardComponents.fla library.
16. Select File > Import > Open External Library and select the DialAssets.fla file.
    - In Windows: C:\Program Files\Macromedia\Flash 8\Samples and Tutorials\Samples\Components\DialComponent\DialAssets.fla.
    - On the Macintosh: HD/Applications/Macromedia Flash 8/Samples and Tutorials/Samples/Components/DialComponent/DialAssets.fla
17. In the Assets layer, select Frame 2 and drag an instance of the DialFinal movie clip from the DialAssets library to the Stage.

    All the component assets are added to Frame 2 of the Assets layer. Because Frame 1 of the Actions layer has a `stop()` global function, the assets in Frame 2 will not be seen as they are arranged on the Stage.

    You add assets to Frame 2 for two reasons:
    - So that all assets and sub assets are automatically copied into the library and are available to instantiate dynamically (in the case of DialFinal), or to access their methods, properties, and events (in the case of UIComponent).
    - Placing assets in a frame ensures that they are loaded more smoothly as the movie is streamed, so you don't need to set the library assets to export in the first frame. This approach prevents an initial spike of data transfer while downloading.
18. Close the DialAssets.fla library.
19. In the Assets layer, select Frame 1. Drag the BoundingBox movie clip from the library (Flash UI Components 2 > Component Assets folder) to the Stage. Name the BoundingBox instance **boundingBox_mc**. Use the Info panel to set both the height and width of the DialFinal movie clip to **250** pixels, and the x, y coordinates at **0**, **0**.

    The BoundingBox instance is used to create the component's live preview and resize during authoring. You must set the size of the bounding box so that it can enclose all the graphical elements in your component.

    NOTE: If you are extending a component (including any version 2 component) you must keep any instance names already in use by that component because its code will refer to those instance names. For example, if you include a version 2 component that is already using the instance name boundingBox_mc, you should not rename it. For your own instance names, you can use any unique name that does not conflict with an existing one within the same scope.
20. Select the Dial movie clip in the library, and select Component Definition from the Library context menu (Windows: Right-click; Mac: control-click).

21. In the AS 2.0 Class text box, enter **Dial**.

    This value is the name of the ActionScript class. If the class is in a package, the value is the full package, for example, mx.controls.CheckBox.

22. Click OK.

23. Save the file.

Creating the Dial class file

Now, you need to create the Dial class file as a new ActionScript file.

To create the Dial class file:

1. In Flash, select File > New and then select ActionScript File.
2. Select File > Save As and save the file as **Dial.as** in the same folder as the Dial.fla file.

NOTE	You can use any text editor to save the Dial.as file.

3. You can copy or type the following Dial component ActionScript class code into your new Dial.as file. Typing rather than copying the code helps you become familiar with each element of the component code.

   Please read the comments in the code for a description of each section. (For detailed information on the elements of a component class file, see "Overview of a component class file" on page 144.

```
// Import the package so you can reference
// the class directly.
import mx.core.UIComponent;

// Event metadata tag
[Event("change")]
class Dial extends UIComponent
{
  // Components must declare these to be proper
  // components in the components framework.
  static var symbolName:String = "Dial";
  static var symbolOwner:Object = Dial;
  var className:String = "Dial";

  // The needle and dial movie clips that are
  // the component's graphical representation
  private var needle:MovieClip;
  private var dial:MovieClip;
  private var boundingBox_mc:MovieClip;
```

```
// The private member variable "__value" is publicly
// accessible through implicit getter/setter methods,
// Updating this property updates the needle's position
// when the value is set.
private var __value:Number = 0;

// This flag is set when the user drags the
// needle with the mouse, and cleared afterwards.
private var dragging:Boolean = false;

// Constructor;
// While required for all classes, v2 components require
// the contstructor to be empty with zero arguments.
// All initialization takes place in a required init()
// method after the class instance has been constructed.
function Dial() {
}

// Initialization code:
// The init() method is required for v2 components. It must also
// in turn call its parent class init() method with super.init().
// The init() method is required for components extending UIComponent.
function init():Void {
  super.init();
  useHandCursor = false;
  boundingBox_mc._visible = false;
  boundingBox_mc._width = 0;
  boundingBox_mc._height = 0;
}
// Create children objects needed at start up:
// The createChildren() method is required for components
// extending UIComponent.
public function createChildren():Void {
  dial = createObject("DialFinal", "dial", 10);
  size();
}

// The draw() method is required for v2 components.
// It is invoked after the component has been
// invalidated by someone calling invalidate().
// This is better than redrawing from within the set() function
// for value, because if there are other properties, it's
// better to batch up the changes into one redraw, rather
// than doing them all individually. This approach leads
// to more efficiency and better centralization of code.
function draw():Void {
  super.draw();
  dial.needle._rotation = value;
}
```

```
// The size() method is invoked when the component's size
// changes. This is an opportunity to resize the children,
// and the dial and needle graphics.
// The size() method is required for components extending UIComponent.
function size():Void {
  super.size();
  dial._width = width;
  dial._height = height;
  // Cause the needle to be redrawn, if necessary.
  invalidate();
}

// This is the getter/setter for the value property.
// The [Inspectable] metadata makes the property appear
// in the Property inspector. This is a getter/setter
// so that you can call invalidate and force the component
// to redraw, when the value is changed.
[Bindable]
[ChangeEvent("change")]
[Inspectable(defaultValue=0)]
function set value (val:Number)
{
  __value = val;
  invalidate();
}

function get value ():Number
{
  return twoDigits(__value);
}

function twoDigits(x:Number):Number
{
  return (Math.round(x * 100) / 100);
}


// Tells the component to expect mouse presses
function onPress()
{
  beginDrag();
}

// When the dial is pressed, the dragging flag is set.
// The mouse events are assigned callback functions.
function beginDrag()
{
  dragging = true;
  onMouseMove = mouseMoveHandler;
  onMouseUp = mouseUpHandler;
```

```
    }

    // Remove the mouse events when the drag is complete
    // and clear the flag.
    function mouseUpHandler()
    {
      dragging = false;
      delete onMouseMove;
      delete onMouseUp;
    }

    function mouseMoveHandler()
    {
      // Calculate the angle
      if (dragging) {
        var x:Number = _xmouse - width/2;
        var y:Number = _ymouse - height/2;

        var oldValue:Number = value;
        var newValue:Number = 90+180/Math.PI*Math.atan2(y, x);
        if (newValue<0) {
          newValue += 360;
        }
        if (oldValue != newValue) {
          value = newValue;
          dispatchEvent( {type:"change"} );
        }
      }
    }
  }
```

Testing and exporting the Dial component

You've created the Flash file that contains the graphical elements, the base classes and the class file that contains all the functionality of the Dial component. Now it's time to test the component.

Ideally, you would test the component as you work, especially while you're writing the class file. The fastest way to test as you work is to convert the component to a compiled clip and use it in the component's FLA file.

When you've completely finished a component, export it as a SWC file. For more information, see "Exporting and distributing a component" on page 182.

To test the Dial component:

1. In the Dial.fla file, select the Dial component in the library, open the Library context menu (Windows: Right-click; Mac: control-click), and select Convert to Compiled Clip.

   A compiled clip is added to the library with the name Dial SWF.

   > NOTE: If you've already created a compiled clip (for example, if this is the second or third time you're testing), a Resolve Library Conflict dialog box appears. Select Replace Existing Items to add the new version to the document.

2. Drag Dial SWF to the Stage on the main Timeline.
3. You can resize it and set its value property in the Property inspector or the Component Inspector. When you set its value property, the needle's position should change accordingly.
4. To test the `value` property at runtime, give the dial the instance name **dial** and add the following code to Frame 1 on the main Timeline:

```
// position of the text field
var textXPos:Number = dial.width/2 + dial.x
var textYPos:Number = dial.height/2 + dial.y;

// creates a text field in which to view the dial.value
createTextField("dialValue", 10, textXPos, textYPos, 100, 20);

// creates a listener to handle the change event
function change(evt){
// places the value property in the text field
// whenever the needle moves
  dialValue.text = dial.value;
}
dial.addEventListener("change", this);
```

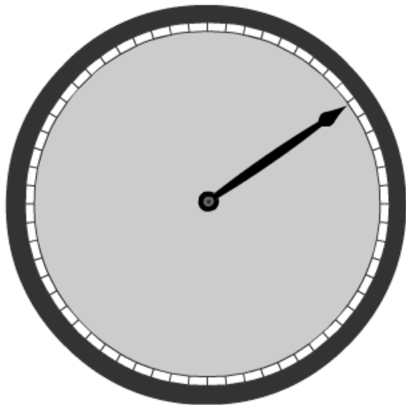
5. Select Control > Test Movie to test the component in Flash Player.

To export the Dial component:

1. In the Dial.fla file, select the Dial component in the library, open the Library context menu (Windows: Right-click; Mac: control-click), and select Export SWC File.
2. Select a location to save the SWC file.

   If you save it to the Components folder in the user-level configuration folder, you can reload the Components panel without restarting Flash and the component appears in the panel.

   > NOTE: For information about folder locations, see "Configuration folders installed with Flash" in *Using Flash*.



The completed Dial component

Selecting a parent class

The first thing to decide when creating a component is whether to extend one of the version 2 classes. If you choose to extend a version 2 class, you can either extend a component class (for example, Button, CheckBox, ComboBox, List, and so on) or one of the base classes, UIObject or UIComponent. All the component classes, except the Media components, extend the base classes; if you extend a component class, the class automatically inherits from the base classes as well.

The two base classes supply common features for components. By extending these classes, your component begins with a basic set of methods, properties, and events.

You don't have to create a subclass UIObject or UIComponent or any other classes in the version 2 framework. Even if your component classes inherit directly from the MovieClip class, you can use many powerful component features: export to a SWC file or compiled clip, use built-in live preview, view inspectable properties, and so on. However, if you want your components to work with the Macromedia version 2 components, and use the manager classes, you need to extend UIObject or UIComponent.

The following table briefly describes the version 2 base classes:

Base class	Extends	Description
`mx.core.UIObject`	MovieClip	UIObject is the base class for all graphical objects. It can have shape, draw itself, and be invisible. UIObject provides the following functionality: • Editing styles • Event handling • Resizing by scaling
`mx.core.UIComponent`	UIObject	UIComponent is the base class for all components. UIComponent provides the following functionality: • Creating focus navigation • Creating a tabbing scheme • Enabling and disabling components • Resizing components • Handling low-level mouse and keyboard events

Understanding the UIObject class

Components based on version 2 of the Macromedia Component Architecture descend from the UIObject class, which is a subclass of the MovieClip class. The MovieClip class is the base class for all classes in Flash that represent visual objects on the screen.

UIObject adds methods that allow you to handle styles and events. It posts events to its listeners just before drawing (the `draw` event is the equivalent of the `MovieClip.onEnterFrame` event), when loading and unloading (`load` and `unload`), when its layout changes (`move`, `resize`), and when it is hidden or revealed (`hide` and `reveal`).

UIObject provides alternate read-only variables for determining the position and size of a component (`width`, `height`, `x`, `y`), and the `move()` and `setSize()` methods to alter the position and size of an object.

The UIObject class implements the following:

- Styles
- Events
- Resize by scaling

Understanding the UIComponent class

The UIComponent class is a subclass of UIObject (see UIComponent class in *Components Language Reference*). It is the base class of all components that handle user interaction (mouse and keyboard input). The UIComponent class allows components to do the following:

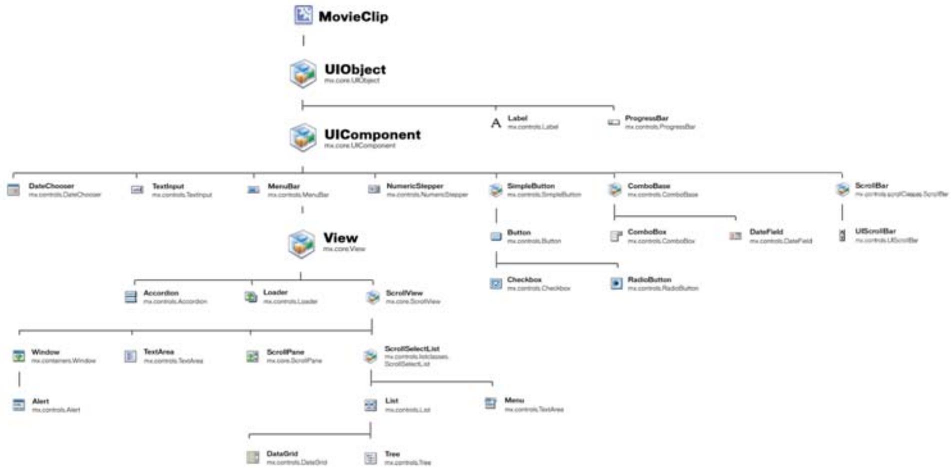
- Receive focus and keyboard input
- Enable and disable components
- Resize by layout

About extending other version 2 classes

To make component construction easier, you can extend any class; you are not required to extend the UIObject or UIComponent class directly. If you extend any other version 2 component's class (except the Media components), you extend UIObject and UIComponent by default. Any component class listed in the Component dictionary can be extended to create a new component class.

For example, if you want to create a component that behaves almost the same as a Button component does, you can extend the Button class instead of re-creating all the functionality of the Button class from the base classes.

The following figure shows the version 2 component hierarchy:



Version 2 component hierarchy

A FlashPaper version of this file is available in the Flash installation directory at this location: Flash 8\Samples and Tutorials\Samples\Components\arch_diagram.swf.

About extending the MovieClip class

You can choose not to extend a version 2 class and have your component inherit directly from the ActionScript MovieClip class. However, if you want any of the UIObject and UIComponent functionality, you'll have to build it yourself. You can open the UIObject and UIComponent classes (First Run/Classes/mx/core) to examine how they are constructed.

Creating a component movie clip

To create a component, you must create a movie clip symbol and link it to the component's class file.

The movie clip has two frames and two layers. The first layer is an Actions layer and has a `stop()` global function on Frame 1. The second layer is an Assets layer with two keyframes. Frame 1 contains a bounding box or any graphics that serve as placeholders for the final art. Frame 2 contains all other assets, including graphics and base classes, used by the component.

Inserting a new movie clip symbol

All components are MovieClip objects. To create a new component, you must first insert a new symbol into a new FLA file.

To add a new component symbol:

1. In Flash, create a blank Flash document.
2. Select Insert > New Symbol.

   The Create New Symbol dialog box appears.
3. Enter a symbol name. Name the component by capitalizing the first letter of each word in the component (for example, MyComponent).
4. Select the Movie Clip behavior.
5. Click the Advanced button to display the advanced settings.
6. Select Export for ActionScript and deselect Export in First Frame and Export in First Frame.
7. Enter a linkage identifier.
8. In the AS 2.0 Class text box, enter the fully qualified path to the ActionScript 2.0 class.

   The class name should be the same as the component name that appears in the Components panel. For example, the Button component's class is mx.controls.Button.

   > NOTE: Do not include the filename's extension; the AS 2.0 Class text box points to the packaged location of the class and not the file system's name for the file.

   If the ActionScript file is in a package, you must include the package name. This value can be relative to the classpath or can be an absolute package path (for example, mypackage.MyComponent).
9. In most cases, you should deselect Export in First Frame (it is selected by default). For more information, see "Component development checklist" on page 186.
10. Click OK.

    Flash adds the symbol to the library and switches to symbol-editing mode. In this mode, the name of the symbol appears above the upper-left corner of the Stage, and a cross hair indicates the symbol's registration point.

Editing the movie clip

After you create the new symbol and define the linkages for it, you can define the component's assets in the symbol's Timeline.

A component's symbol should have two layers. This section describes what layers to insert and what to add to those layers.

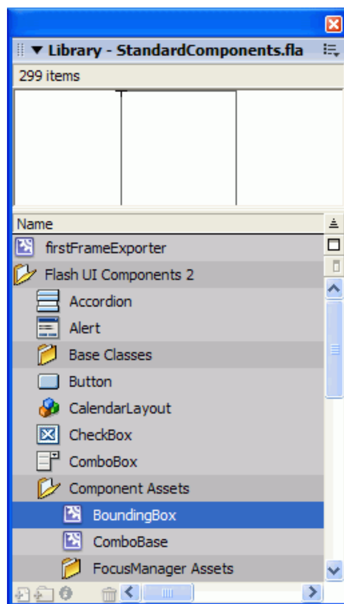
To edit the movie clip:

1. Rename Layer 1 **Actions** and select Frame 1.
2. Open the Actions panel and add a `stop()` function, as follows:

   ```
   stop();
   ```

   Do not add any graphical assets to this frame.
3. Add a Layer named **Assets**.
4. On the Assets layer, select Frame 2 and insert a blank keyframe.

   There are now two blank keyframes in this layer.
5. Do one of the following:
   - If the component has visual assets that define the bounding area, drag the symbols into Frame 1 and arrange them appropriately.
   - If your component creates all its visual assets at runtime, drag a BoundingBox symbol to the Stage in Frame 1, size it correctly, and name the instance **boundingBox_mc**. The symbol is located in the library of the StandardComponents.fla that is located in the Configuration/ComponentFLA folder.



6. If you are extending an existing component, place an instance of that component and any other base classes in Frame 2 of the Assets layer.

   To do this, select the symbol from the Components panel and drag it to the Stage. If you are extending a base class, open StandardComponents.fla from the Configuration/ComponentFLA folder and drag the class from the library to the Stage.

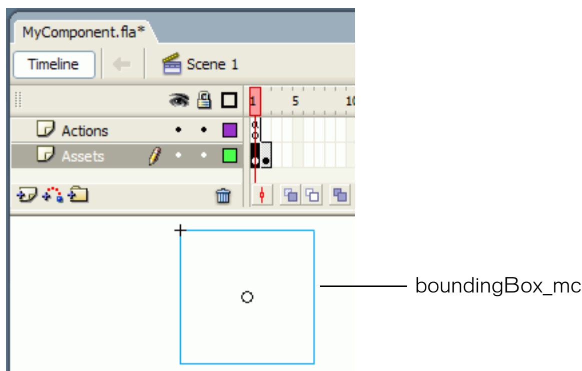
   NOTE

   When you drag UIComponent to the component library, the folder hierarchy in the library is changed. If you plan to use your library again, or use it with other groups of components (such as the version 2 components), you should restructure the folder hierarchy to match the StandardComponents.fla library so that it's organized well and you avoid duplicate symbols.

7. Add any graphical assets used by this component on Frame 2 of your component's Assets layer.

   Any asset that a component uses (whether it's another component or media such as bitmaps) should have an instance placed in the second frame of the Assets layer.

8. Your finished symbol should look something like this:



Defining the movie clip as a component

The movie clip symbol must be linked to an ActionScript class file in the Component Definition dialog box. This is how Flash knows where to look for the component's metatags. (For more information about metatags, see "Adding component metadata" on page 149.) There are other options you can select in the Component Definition dialog box as well.

To define a movie clip as a component:

1. Select the movie clip in the library and select Component Definition from the Library context menu (Windows: Right-click; Mac: control-click).
2. You must enter an AS 2.0 class.

   If the class is part of a package, enter the full package name.
3. Specify other options in the Component Definition dialog box, if desired:
   - Click the Plus (+) button to define parameters.

     This is optional. The best practice is to use the metadata Inspectable tag in the component's class file to specify parameters. When an ActionScript 2.0 class is not specified, define the parameters for the component here.
   - Specify a custom UI.

     This is a SWF file that plays in the Component inspector. You can embed it in the component FLA file or browse to an external SWF.
   - Specify a live preview.

     This is an external or embedded SWF file. You don't need to specify a live preview here; you can add a bounding box to the component movie clip, and Flash creates a live preview for you. See "Creating a component movie clip" on page 138.
   - Enter a description.

     The Description field was deprecated in Flash MX 2004 because the Reference panel has been removed. This field is provided for backward compatibility when you save FLA files in the Flash MX format.
   - Choose an icon.

     This option specifies a PNG file to use as an icon for the component. If you specify an IconFile metadata tag in the ActionScript 2.0 class file (best practice), this field is ignored.
   - Select or deselect Parameters Are Locked in Instances.

     When this option is unselected, users can add parameters to each component instance that differ from the component's parameters. Generally, this setting should be selected. This option provides backward compatibility with Flash MX.
   - Specify a tooltip that appears in the Components panel.

Creating the ActionScript class file

All component symbols are linked to an ActionScript 2.0 class file. (For information on linking, see "Creating a component movie clip" on page 138.)

To edit ActionScript class files, you can use Flash, any text editor, or any Integrated Development Environment (IDE).

The external ActionScript class extends another class (whether the class is a version 2 component, a version 2 base class, or the ActionScript MovieClip class). You should extend the class that creates the functionality that is most similar to the component you want to create. You can inherit from (extend) only one class. ActionScript 2.0 does not allow multiple inheritance.

Simple example of a component class file

The following is a simple example of a class file called MyComponent.as. If you were creating this component, you would link this file to the component movie clip in Flash.

This example contains a minimal set of imports, methods, and declarations for a component, MyComponent, that inherits from the UIComponent class. The MyComponents.as file is saved in the myPackage folder.

```
[Event("eventName")]

// Import packages.
import mx.core.UIObject;

// Declare the class and extend from the parent class.
class mypackage.MyComponent extends UIObject {

  // Identify the symbol name that this class is bound to.
  static var symbolName:String = "mypackage.MyComponent";

  // Identify the fully qualified package name of the symbol owner.
  static var symbolOwner:Object = Object(mypackage.MyComponent);

  // Provide the className variable.
  var className:String = "MyComponent";

  // Define an empty constructor.
  function MyComponent() {
  }

  // Call the parent's init() method.
  // Hide the bounding box--it's used
  // only during authoring.
```

```
  function init():Void {
    super.init();

     boundingBox_mc.width = 0;
     boundingBox_mc.height = 0;
     boundingBox_mc.visible = false;

  }

  function createChildren():Void{
    // Call createClassObject to create subobjects.
    size();
    invalidate();
  }

  function size(){
    // Write code to handle sizing.
    super.size();
    invalidate();
  }

  function draw(){
    // Write code to handle visual representation.
    super.draw();
  }

}
```

Overview of a component class file

The following is a general procedure that describes how to create an ActionScript file for a component class. Some steps may be optional, depending on the type of component you create.

To write a component class file:

1. (Optional) Import classes. (See "Importing classes" on page 146).

   This allows you to refer to classes without writing out the package (for example, Button instead of mx.controls.Button).

2. Define the class using the `class` keyword; use the `extend` keyword to extend a parent class. (See "Defining the class and its superclass" on page 146).

3. Define the `symbolName`, `symbolOwner`, and `className` variables. (See "Identifying the class, symbol, and owner names" on page 147).

   These variables are necessary only in version 2 components.

4. Define member variables. (See "Defining variables" on page 148).

   These can be used in getter/setter methods.

5. Define a constructor function. (See "About the constructor function" on page 164).

6. Define an `init()` method. (See "Defining the init() method" on page 162).

   This method is called when the class is created if the class extends UIComponent. If the class extends MovieClip, call this method from the constructor function.

7. Define a `createChildren()` method. (See "Defining the createChildren() method" on page 162).

   This method is called when the class is created if the class extends UIComponent. If the class extends MovieClip, call this method from the constructor function.

8. Define a `size()` method. (See "Defining the size() method" on page 165).

   This method is called when the component is resized, if the class extends UIComponent. In addition, this method is called when the component's live preview is resized during authoring.

9. Define a `draw()` method. (See "About invalidation" on page 166).

   This method is called when the component is invalidated, if the class extends UIComponent.

10. Add a Metadata tag and declaration. (See "Adding component metadata" on page 149).

    Adding the tag and declaration causes getter/setter properties to appear in the Property inspector and Component inspector in Flash.

11. Define getter/setter methods. (See "Using getter/setter methods to define parameters" on page 148).

12. (Optional) Create variables for every skin element/linkage used in the component. (See "About assigning skins" on page 168).

    This allows users to set a different skin element by changing a parameter in the component.

Importing classes

You can import class files so that you don't have to write fully qualified class names throughout your code. This can make code more concise and easier to read. To import a class, use the `import` statement at the top of the class file, as in the following:

```
import mx.core.UIObject;
import mx.core.ScrollView;
import mx.core.ext.UIObjectExtensions;

class MyComponent extends UIComponent{
```

You can also use the wildcard character (*) to import all the classes in a given package. For example, the following statement imports all classes in the `mx.core` package:

```
import mx.core.*;
```

If an imported class is not used in a script, the class is not included in the resulting SWF file's bytecode. As a result, importing an entire package with a wildcard does not create an unnecessarily large SWF file.

Defining the class and its superclass

A component class file is defined like any class file. Use the `class` keyword to indicate the class name. The class name must also be the name of the class file. Use the `extends` keyword to indicate the superclass. For more information, see Chapter 7, "Writing custom class files," in *Learning ActionScript 2.0 in Flash*.

```
class MyComponentName extends UIComponent{

}
```

Identifying the class, symbol, and owner names

To help Flash find the proper ActionScript classes and packages and to preserve the component's naming, you must set the `symbolName`, `symbolOwner`, and `className` variables in your component's ActionScript class file.

The `symbolOwner` variable is an Object reference that refers to a symbol. If the component is its own `symbolOwner` or is the `symbolOwner` has been imported, it does not have to be fully qualified.

The following table describes these variables:

Variable	Type	Description
`symbolName`	String	The name of the ActionScript class (for example, ComboBox). This name must match the symbol's linkage identifier. This variable must be static.
`symbolOwner`	Object	The fully qualified class name (for example, mypackage.MyComponent). Do not use quotation marks around the `symbolOwner` value, because it is an Object data type. This name must match the AS 2.0 class in the Linkage Properties dialog box. This variable is used in the internal call to the `createClassObject()` method. This variable must be static.
`className`	String	The name of the component class. This does not include the package name and has no corresponding setting in the Flash development environment. You can use the value of this variable when setting style properties.

The following example adds the `symbolName`, `symbolOwner`, and `className` variables to the MyButton class:

```
class MyButton extends mx.controls.Button {
  static var symbolName:String = "MyButton";
  static var symbolOwner = myPackage.MyButton;
  var className:String = "MyButton";
}
```

Defining variables

The following code sample is from Button.as file (mx.controls.Button). It defines a variable, `btnOffset`, to use in the class file. It also defines the variables `__label`, and `__labelPlacement`. The latter two variables are prefixed with a double underscore to prevent name conflicts when they are used in getter/setter methods, and ultimately used as properties and parameters in the component. For more information, see "Using getter/setter methods to define parameters" on page 148 in *Learning ActionScript 2.0 in Flash*.

```
/**
* Number used to offset the label and/or icon when button is pressed.
*/
  var btnOffset:Number = 0;

/**
*@private
* Text that appears in the label if no value is specified.
*/
  var __label:String = "default value";

/**
*@private
* default label placement
*/
  var __labelPlacement:String = "right";
```

Using getter/setter methods to define parameters

The simplest way to define a component parameter is to add a public member variable that makes the parameter "inspectable." You can do this by using the Inspectable tag in the Component inspector, or adding the Inspectable variable as follows:

```
[Inspectable(defaultValue="strawberry")]
public var flavorStr:String;
```

However, if code that employs a component modifies the `flavorStr` property, the component typically must perform an action to update itself in response to the property change. For example, if `flavorStr` is set to `"cherry"`, the component might redraw itself with a cherry image instead of the default strawberry image.

For regular member variables, the component is not automatically notified that the member variable's value has changed.

Getter/setter methods are a straightforward way to detect changes to component properties. Instead of declaring a regular variable with `var`, declare getter/setter methods, as follows:

```
private var __flavorStr:String = "strawberry";

[Inspectable(defaultValue="strawberry")]

public function get flavorStr():String{
  return __flavorStr;
}
public function set flavorStr(newFlavor:String) {
  __flavorStr = newFlavor;
  invalidate();
}
```

The `invalidate()` call causes the component to redraw itself with the new flavor. This is the benefit of using getter/setter methods for the `flavorStr` property, instead of a regular member variable. See "Defining the draw() method" on page 165.

To define getter/setter methods, remember these points:

- Precede the method name with `get` or `set`, followed by a space and the property name.
- The variable that stores the property's value cannot have the same name as the getter or setter. By convention, precede the names of the getter and setter variables with two underscores.

Getters and setters are commonly used in conjunction with tags to define properties that are visible in the Property and Component inspectors.

For more information about getter/setter methods, see "About getter and setter methods" in *Learning ActionScript 2.0 in Flash*.

Adding component metadata

You can add component metadata tags in your external ActionScript class files to tell the compiler about component parameters, data binding properties, and events. Metadata tags are used in the Flash authoring environment for a variety of purposes.

The metadata tags can only be used in external ActionScript class files. You cannot use metadata tags in FLA files.

Metadata is associated with a class declaration or an individual data field. If the value of an attribute is a string, you must enclose that attribute in quotation marks.

Metadata statements are bound to the next line of the ActionScript file. When defining a component property, add the metadata tag on the line before the property declaration. The only exception is the Event metadata tag. When defining component events, add the metadata tag outside the class definition so that the event is bound to the entire class.

In the following example, the Inspectable tags define the `flavorStr`, `colorStr`, and `shapeStr` parameters:

```
[Inspectable(defaultValue="strawberry")]
public var flavorStr:String;
[Inspectable(defaultValue="blue")]
public var colorStr:String;
[Inspectable(defaultValue="circular")]
public var shapeStr:String;
```

In the Property inspector and the Parameters tab of the Component inspector, Flash displays all of these parameters as type String.

Metadata tags

The following table describes the metadata tags you can use in ActionScript class files:

Tag	Description
Inspectable	Exposes a property in the Component inspector and Property inspector. See "About the Inspectable tag" on page 151.
InspectableList	Identifies which subset of inspectable properties should be listed in the Property inspector and Component inspector. If you don't add an InspectableList attribute to your component's class, all inspectable parameters appear in the Property inspector. See "About the InspectableList tag" on page 153.
Event	Defines a component event. See "About the Event tag" on page 153.
Bindable	Reveals a property in the Bindings tab of the Component inspector. See "About the Bindable tag" on page 154.
ChangeEvent	Identifies a event that cause data binding to occur. See "About the ChangeEvent tag" on page 156.
Collection	Identifies a `collection` attribute exposed in the Component inspector. See "About the Collection tag" on page 157.
IconFile	Specifies the filename for the icon that represents this component in the Components panel. See "About the IconFile tag" on page 158.
ComponentTask	Specifies the filenames of one or more associated JSFL files to perform tasks in the authoring environment. See "About the ComponentTask tag" on page 158.

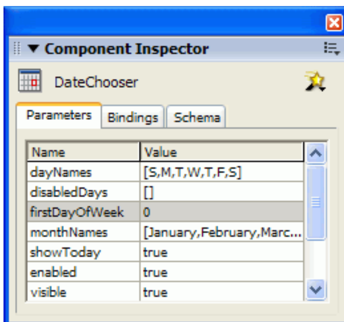
The following sections describe the component metadata tags in more detail.

About the Inspectable tag

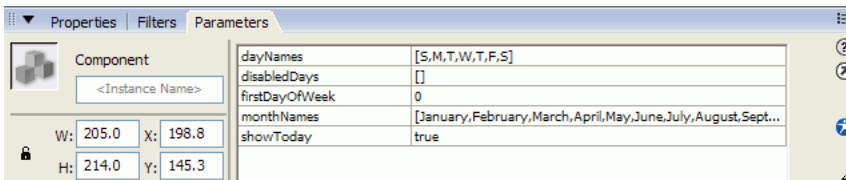
You use the Inspectable tag to specify the user-editable (inspectable) parameters that appear in the Component inspector and Property inspector. This lets you maintain the inspectable properties and the underlying ActionScript code in the same place. To see the component properties, drag an instance of the component onto the Stage and select the Parameters tab of the Component inspector.

Collection parameters are also inspectable. For more information, see “About the Collection tag” on page 157.

The following figure shows the Parameters tab of the Component inspector for the DateChooser component:



Alternatively, you can view a subset of the component properties on the Property inspector Parameters tab.



When determining which parameters to reveal in the authoring environment, Flash uses the Inspectable tag. The syntax for this tag is as follows:

```
[Inspectable(value_type=value[,attribute=value,...])]
property_declaration name:type;
```

The following example defines the `enabled` parameter as inspectable:

```
[Inspectable(defaultValue=true, verbose=1, category="Other")]
var enabled:Boolean;
```

The Inspectable tag also supports loosely typed attributes like this:

```
[Inspectable("danger", 1, true, maybe)]
```

The metadata statement must immediately precede the property's variable declaration in order to be bound to that property.

The following table describes the attributes of the Inspectable tag:

Attribute	Type	Description
`defaultValue`	String or Number	(Optional) A default value for the inspectable property.
`enumeration`	String	(Optional) Specifies a comma-delimited list of legal values for the property.
`listOffset`	Number	(Optional) Added for backward compatibility with Flash MX components. Used as the default index into a List value.
`name`	String	(Optional) A display name for the property. For example, Font Width. If not specified, use the property's name, such as `_fontWidth`.
`type`	String	(Optional) A type specifier. If omitted, use the property's type. The following values are acceptable: • Array • Boolean • Color • Font Name • List • Number • Object • String
`variable`	String	(Optional) Added for backward compatibility with Flash MX components. Specifies the variable that this parameter is bound to.
`verbose`	Number	(Optional) An inspectable property that has the verbose attribute set to 1 does not appear in the Property inspector but does appear in the Component inspector. This is typically used for properties that are not modified frequently.

None of these attributes are required; you can use Inspectable as the metadata tag.

All properties of the superclass that are marked Inspectable are automatically inspectable in the current class. Use the InspectableList tag if you want to hide some of these properties for the current class.

About the InspectableList tag

Use the InspectableList tag to specify which subset of inspectable properties should appear in the Property inspector. Use InspectableList in combination with Inspectable so that you can hide inherited attributes for components that are subclasses. If you do not add an InspectableList tag to your component's class, all inspectable parameters, including those of the component's parent classes, appear in the Property inspector.

The InspectableList syntax is as follows:

```
[InspectableList("attribute1"[,...])]
// class definition
```

The InspectableList tag must immediately precede the class definition because it applies to the entire class.

The following example allows the `flavorStr` and `colorStr` properties to be displayed in the Property inspector, but excludes other inspectable properties from the Parent class:

```
[InspectableList("flavorStr","colorStr")]
class BlackDot extends DotParent {
  [Inspectable(defaultValue="strawberry")]
  public var flavorStr:String;
  [Inspectable(defaultValue="blue")]
  public var colorStr:String;
  ...
}
```

About the Event tag

Use the Event tag to define events that the component emits.

This tag has the following syntax:

```
[Event("event_name")]
```

For example, the following code defines a `click` event:

```
[Event("click")]
```

Add the Event statements outside the class definition in the ActionScript file so that the events are bound to the class and not a particular member of the class.

The following example shows the Event metadata for the UIObject class, which handles the `resize`, `move`, and `draw` events:

```
...
import mx.events.UIEvent;
[Event("resize")]
[Event("move")]
[Event("draw")]
class mx.core.UIObject extends MovieClip {
  ...
}
```

To broadcast a particular instance, call the `dispatchEvent()` method. See "Using the dispatchEvent() method" on page 166.

About the Bindable tag

Data binding connects components to each other. You achieve visual data binding through the Bindings tab of the Component inspector. From there, you add, view, and remove bindings for a component.

Although data binding works with any component, its main purpose is to connect user interface components to external data sources, such as web services and XML documents. These data sources are available as components with properties, which you can bind to other component properties.

Use the Bindable tag before a property in an ActionScript class to make the property appear in the Bindings tab in the Component inspector. You can declare a property by using `var` or getter/setter methods. If a property has both a getter and a setter method, you only need to apply the Bindable tag to one.

The Bindable tag has the following syntax:

```
[Bindable "readonly"|"writeonly",type="datatype"]
```

Both attributes are optional.

The following example defines the variable `flavorStr` as a property that is accessible on the Bindings tab of the Component inspector:

```
[Bindable]
public var flavorStr:String = "strawberry";
```

The Bindable tag takes three options that specify the type of access to the property, as well as the data type of that property. The following table describes these options:

Option	Description
`readonly`	Indicates that when you create bindings in the Component inspector, you can only create bindings that use this property as a source. However, if you use ActionScript to create bindings, there is no such restriction. `[Bindable("readonly")]`
`writeonly`	Indicates that when you create bindings in the Component inspector, this property can only be used as the destination of a binding. However, if you use ActionScript to create bindings, there is no such restriction. `[Bindable("writeonly")]`

`type="`*`datatype`*`"`	Indicates the type that data binding uses for the property. The rest of Flash uses the declared type.
	If you do not specify this option, data binding uses the property's data type as declared in the ActionScript code.
	In the following example, data binding will treat `x` as type DataProvider, even though it is really type Object:
	`[Bindable(type="DataProvider")]`
	`var x: Object;`

All properties of all components can participate in data binding. The Bindable tag merely controls which of those properties are available for binding in the Component inspector. If a property is not preceded by the Bindable tag, you can still use it for data binding, but you have to create the bindings using ActionScript.

The Bindable tag is required when you use the ChangeEvent tag.

For information on creating data binding in the Flash authoring environment, see "Data binding (Flash Professional only)" in *Using Flash*.

About the ChangeEvent tag

The ChangeEvent tag tells data binding that the component will generate an event any time the value of a specific property changes. In response to the event, data binding executes any binding that has that property as a source. The component only generates the event if you write appropriate ActionScript code in the component. The event should be included in the list of Event metadata declared by the class.

You can declare a property by using `var` or getter/setter methods. If a property has both a getter and a setter method, you only need to apply the ChangeEvent tag to one.

The ChangeEvent tag has the following syntax:

```
[Bindable]
[ChangeEvent("event")]
property_declaration or getter/setter function
```

In the following example, the component generates the `change` event when the value of the bindable property `flavorStr` changes:

```
[Bindable]
[ChangeEvent("change")]
public var flavorStr:String;
```

When the event that is specified in the metadata occurs, Flash notifies bindings that the property has changed.

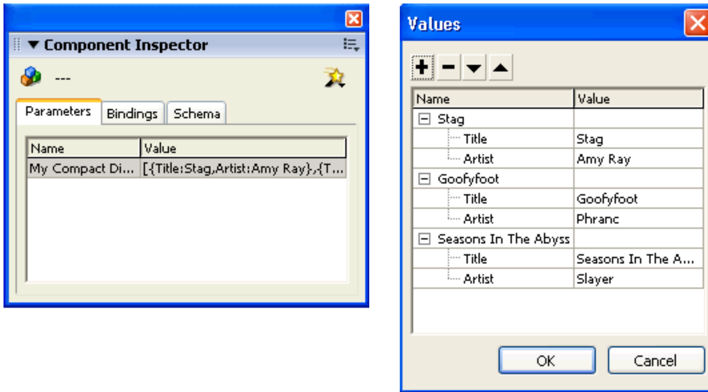
You can register multiple events in the tag, as the following example shows:

```
[ChangeEvent("change1", "change2", "change3")]
```

Any one of those events indicates a change to the property. They do not all have to occur to indicate a change.

About the Collection tag

Use the Collection tag to describe an array of objects that can be modified as a collection of items in the Values dialog box while authoring. The type of the objects is identified by the `collectionItem` attribute. A collection property contains a series of collection items that you define in a separate class. This class is either mx.utils.CollectionImpl or a subclass of it. The individual objects are accessed through the methods of the class identified by the `collectionClass` attribute.



A collection property in the Component inspector and the Values dialog box that appears when you click the magnifying glass.

The syntax for the Collection tag is as follows:

```
[Collection (name="name", variable="varname",
  collectionClass="mx.utils.CollectionImpl",
  collectionItem="coll-item-classname", identifier="string")]
public var varname:mx.utils.Collection;
```

The following table describes the attributes of the Collection tag:

Attribute	Type	Description
`name`	String	(Required) Name that appears in the Component inspector for the collection.
`variable`	String	(Required) ActionScript variable that points to the underlying Collection object (for example, you might name a `Collection` parameter Columns, but the underlying `variable` attribute might be __columns).
`collectionClass`	String	(Required) Specifies the class type to be instantiated for the collection property. This is usually `mx.utils.CollectionImpl`, but it can also be a class that extends `mx.utils.CollectionImpl`.
`collectionItem`	String	(Required) Specifies the class of the collection items to be stored within the collection. This class includes its own inspectable properties that are exposed through metadata.
`identifier`	String	(Required) Specifies the name of an inspectable property within the collection item class that Flash MX uses as the default identifier when the user adds a new collection item through the Values dialog box. Each time a user creates a new collection item, Flash MX sets the item name to *identifier* plus a unique index (for example, if `identifier=name`, the Values dialog box displays name0, name1, name2, and so on).

For more information, see "Collection Properties" on page 187.

About the IconFile tag

You can add an icon that represents your component in the Components panel of the Flash authoring environment. For more information, see "Adding an icon" on page 185.

About the ComponentTask tag

You can specify one or more Flash JavaScript (JSFL) files to perform tasks for your component from within the Flash authoring environment. Use the ComponentTask tag to define this association between your component and its JSFL file and to associate any additional files required a JSFL file. The JSFL files interact with the JavaScript API in the Macromedia Flash authoring environment.

NOTE Any JSFL task files and required dependency files declared with the ComponentTask tag must reside in the same folder as your component FLA file when you export your component as a SWC file.

The ComponentTask tag has the following syntax:

```
[ComponentTask [taskName,taskFile [,otherFile[,…]]]
```

The `taskName` and `taskFile` attributes are required. The `otherFile` attribute is optional

The following example associates `SetUp.jsfl` and `AddNewSymbol.jsfl` with the component class named myComponent. The `AddNewSymbol.jsfl` requires a `testXML.xml` file and is specified in the `otherFile` attribute.

```
[ComponentTask("Do Some Setup","SetUp.jsfl")]
[ComponentTask("Add a new Symbol","AddNewSymbol.jsfl","testXML.xml")]
class myComponent{
  //...
}
```

The following table describes the attributes of the ComponentTask tag:

Attribute	Type	Description
`taskName`	String	(Required) The name of the task as a string. This name is displayed in the Tasks pop-up menu that appears on the Schema tab of the Component inspector.
`taskFile`	String	(Required) The name of the JSFL file that implements the tasks within the authoring environment. The file must reside in the same folder as your component FLA when you export your component as a SWC file.
`otherFile`	String	(Optional) One or more names of files that are required by the JSFL file such as an XML file. The file(s) must reside in the same folder as your component FLA when you export your component as a SWC file.

Defining component parameters

When building a component, you can add parameters that define its appearance and behavior. The most commonly used parameters appear as authoring parameters in the Component inspector and Property inspector. You can also set all inspectable and collection parameters with ActionScript. You define these properties in the component class file by using the Inspectable tag (see “About the Inspectable tag” on page 151).

The following example sets several component parameters in the JellyBean class file, and exposes them with the Inspectable tag in the Component inspector:

```
class JellyBean{
  // a string parameter
  [Inspectable(defaultValue="strawberry")]
  public var flavorStr:String;

  // a string list parameter

  [Inspectable(enumeration="sour,sweet,juicy,rotten",defaultValue="sweet")
  ]
  public var flavorType:String;

  // an array parameter
  [Inspectable(name="Flavors", defaultValue="strawberry,grape,orange",
  verbose=1, category="Fruits")]
  var flavorList:Array;

  // an object parameter
  [Inspectable(defaultValue="belly:flop,jelly:drop")]
  public var jellyObject:Object;

  // a color parameter
  [Inspectable(defaultValue="#ffffff")]
  public var jellyColor:Color;

  // a setter
  [Inspectable(defaultValue="default text")]
  function set text(t:String)

}
```

You can use any of the following types for parameters:

- Array
- Object
- List
- String
- Number
- Boolean
- Font Name
- Color

> **NOTE** The JellyBean class is a theoretical example. To see an actual example, look at the Button.as class file that installs with Flash in the *language*/First Run/Classes/mx/controls directory.

About core functions

You must define five functions in the component class file: `init()`, `createChildren()`, the constructor function, `draw()`, and `size()`. When a component extends the UIComponent, these five functions in the class file are called in the following order:

- `init()`

  Any initialization occurs during the init() function call. For example, instance member variables can be set at this time and the component bounding box can be hidden.

  After `init()` is called, the `width` and `height` properties are automatically set. See "Defining the init() method" on page 162.
- `createChildren()`

  Called as a frame plays in the Timeline. During this time, the component user can call methods and properties to set up the component. Any subobjects the component needs to create are created within the `createChildren()` function. See "Defining the createChildren() method" on page 162.
- Constructor function

  Called to create an instance of the component. The component constructor function is generally left empty to avoid initialization conflicts. See "About the constructor function" on page 164.
- `draw()`

  Any visual elements of the component that are programmatically created or modified should occur within the draw function. See "Defining the draw() method" on page 165.
- `size()`

  This function is called whenever a component is resized at runtime and is passed updated `width` and `height` properties of the component. Component subobjects can be sized or moved in relation to the component's updated `width` and `height` properties within the size() function. See "Defining the size() method" on page 165.

These core component functions are described in detail in the sections that follow.

Defining the init() method

Flash calls the `init()` method when the class is created. This method is called only once when a component is instantiated.

You should use the `init()` method to do the following:

- Call `super.init()`.

  This is required.

- Make the `boundingBox_mc` invisible.

  ```
  boundingBox_mc.width = 0;
  boundingBox_mc.height = 0;
  boundingBox_mc.visible = false;
  ```

- Create instance member variables.

The `width`, `height`, and `clip` parameters are properly set only after this method is called.

The `init()` method is called from UIObject's constructor, so the flow of control climbs up the chain of constructors until it reaches UIObject. UIObject's constructor calls the `init()` method that is defined on lowest subclass. Each implementation of `init()` should call `super.init()` so that its base class can finish initializing. If you implement an `init()` method and you don't call `super.init()`, the `()init` method is not called on any of the base classes, so they might never be in a usable state.

Defining the createChildren() method

Components implement the `createChildren()` method to create subobjects (such as other components) in the component. Rather than calling the subobject's constructor in the `createChildren()` method, call `createClassObject()` or `createObject()` to instantiate a subobject of your component.

It's a good idea to call `size()` within the `createChildren()` method to make sure all children are set to the correct size initially. Also, call `invalidate()` within the `createChildren()` method to refresh the screen. (For more information, see "About invalidation" on page 166.)

The `createClassObject()` method has the following syntax:

```
createClassObject(className, instanceName, depth, initObject)
```

The following table describes the parameters:

Parameter	Type	Description
className	Object	The name of the class.
instanceName	String	The name of the instance.
depth	Number	The depth for the instance.
initObject	Object	An object that contains initialization properties.

To call `createClassObject()`, you must know what the children are, because you must specify the name and type of the object, plus any initialization parameters in the call to `createClassObject()`.

The following example calls `createClassObject()` to create a new Button object for use inside a component:

```
up_mc.createClassObject(mx.controls.Button, "submit_btn", 1);
```

You set properties in the call to `createClassObject()` by adding them as part of the *`initObject`* parameter. The following example sets the value of the `label` property:

```
form.createClassObject(mx.controls.CheckBox, "cb", 0, {label:"Check
  this"});
```

The following example creates TextInput and SimpleButton components:

```
function createChildren():Void {
  if (text_mc == undefined)
    createClassObject(TextInput, "text_mc", 0, { preferredWidth: 80,
  editable:false });
    text_mc.addEventListener("change", this);
    text_mc.addEventListener("focusOut", this);

  if (mode_mc == undefined)
    createClassObject(SimpleButton, "mode_mc", 1, { falseUpSkin:
  modeUpSkinName, falseOverSkin: modeOverSkinName, falseDownSkin:
  modeDownSkinName });
  mode_mc.addEventListener("click", this);
  size()
  invalidate()
}
```

About the constructor function

You can recognize a constructor function because it has the same name as the component class. For example, the following code shows the ScrollBar component's constructor function:

```
function ScrollBar() {
}
```

In this case, when a new scroll bar is instantiated, the `ScrollBar()` constructor is called.

Generally, component constructors should be empty. Setting properties in constructors can sometimes lead to overwriting default values, depending on the order of initialization calls.

If your component extends UIComponent or UIObject, Flash automatically calls `init()`, `createChildren()`, and `size()` methods and you can leave your constructor function empty, as shown here:

```
class MyComponent extends UIComponent{
  ...
  // this is the constructor function
  function MyComponent(){
  }
}
```

All version 2 components should define an `init()` function that is called after the constructor has been called. You should place the initialization code in the component's `init()` function. For more information, see the next section.

If your component extends MovieClip, you may want to call an `init()` method, a `createChildren()` method, and a method that lays out your component from the constructor function, as shown in the following code example:

```
class MyComponent extends MovieClip{
  ...
  function MyComponent(){
    init()
  }

  function init():Void{
    createChildren();
    layout();
  }
  ...
}
```

For more information about constructors, see "Writing the constructor function" in *Learning ActionScript 2.0 in Flash*.

Defining the draw() method

You can write code in the `draw()` method to create or modify visual elements of a component. In other words, in the `draw()` method, a component draws itself to match its state variables. Since the last `draw()` method was called, multiple properties or methods may have been called, and you should try to account for all of them in the body of `draw()`.

However, you should not call the `draw()` method directly. Instead, call the `invalidate()` method so that calls to `draw()` can be queued and handled in a batch. This approach increases efficiency and centralizes code. (For more information, see "About invalidation" on page 166.)

Inside the `draw()` method, you can use calls to the Flash drawing API to draw borders, rules, and other graphical elements. You can also set property values and call methods. You can also call the `clear()` method, which removes the visible objects.

In the following example from the Dial component (see "Building your first component" on page 127), the `draw()` method sets the rotation of the needle to the `value` property:

```
function draw():Void {
  super.draw();
  dial.needle._rotation = value;
}
```

Defining the size() method

When a component is resized at runtime using the `componentInstance.setSize()` method, the `size()` function is invoked and passed `width` and `height` properties. You can use the `size()` method in the component's class file to lay out the contents of the component.

At a minimum, the `size()` method should call the superclass's `size()` method (`super.size()`).

In the following example from the Dial component (see "Building your first component" on page 127), the `size()` method uses the `width` and `height` parameters to resize the dial movie clip:

```
function size():Void {
  super.size();
  dial._width = width;
  dial._height = height;
  invalidate();
}
```

Call the `invalidate()` method inside the `size()` method to tag the component for redraw instead of calling the `draw()` method directly. For more information, see the next section.

About invalidation

Macromedia recommends that a component not update itself immediately in most cases, but that it instead should save a copy of the new property value, set a flag indicating what is changed, and call the `invalidate()` method. (This method indicates that just the visuals for the object have changed, but size and position of subobjects have not changed. This method calls the `draw()` method.)

You must call an invalidation method at least once during the instantiation of your component. The most common place for you to do this is in the `createChildren()` or `layoutChildren()` methods.

Dispatching events

If you want your component to broadcast events other than the events it may inherit from a parent class, you must call the `dispatchEvent()` method in the component's class file.

The `dispatchEvent()` method is defined in the mx.events.EventDispatcher class and is inherited by all components that extend UIObject. (See "EventDispatcher class" in *Components Language Reference*.)

You should also add an Event metadata tag at the top of the class file for each new event. For more information, see "About the Event tag" on page 153.

NOTE: For information about handling component events in a Flash application, see Chapter 4, "Handling Component Events," on page 63.

Using the dispatchEvent() method

In the body of your component's ActionScript class file, you broadcast events using the `dispatchEvent()` method. The `dispatchEvent()` method has the following syntax:

```
dispatchEvent(eventObj)
```

The *eventObj* parameter is an ActionScript object that describes the event (see the example later in this section).

You must declare the `dispatchEvent()` method in your code before you call it, as follows:

```
private var dispatchEvent:Function;
```

You must also create an event object to pass to `dispatchEvent()`. The event object contains information about the event that the listener can use to handler the event.

You can explicitly build an event object before dispatching the event, as the following example shows:

```
var eventObj = new Object();
eventObj.type = "myEvent";
eventObj.target = this;
dispatchEvent(eventObj);
```

You can also use a shortcut syntax that sets the value of the `type` property and the `target` property and dispatches the event in a single line:

```
ancestorSlide.dispatchEvent({type:"revealChild", target:this});
```

In the preceding example, setting the `target` property is optional, because it is implicit.

The description of each event in the Flash 8 documentation lists the event properties that are optional and required. For example, the `ScrollBar.scroll` event takes a `detail` property in addition to the `type` and `target` properties. For more information, see the event descriptions in *Components Language Reference*.

Common events

The following table lists the common events that are broadcast by various classes. Every component should broadcast these events if they make sense for that component. This is not a complete list of events for all components, just ones that are likely to be reused by other components. Even though some events specify no parameters, all events have an implicit parameter: a reference to the object broadcasting the event.

Event	Use
`click`	Used by the Button component, or whenever a mouse click has no other meaning.
`change`	Used by List, ComboBox, and other text-entry components.
`scroll`	Used by ScrollBar and other controls that cause scrolling (scroll “bumpers” on a scrolling pop-up menu).

In addition, because of inheritance from the base classes, all components broadcast the following events:

UIComponent event	Description
`load`	The component is creating or loading its subobjects.
`unload`	The component is unloading its subobjects.
`focusIn`	The component now has the input focus. Some HTML-equivalent components (ListBox, ComboBox, Button, Text) might also broadcast `focus`, but all broadcast `DOMFocusIn`.
`focusOut`	The component has lost the input focus.
`move`	The component has been moved to a new location.
`resize`	The component has been resized.

The following table describes common key events:

Key events	Description
`keyDown`	A key is pressed. The `code` property contains the key code and the `ascii` property contains the ASCII code of the key pressed. Do not check with the low-level Key object, because the event might not have been generated by the Key object.
`keyUp`	A key is released.

About assigning skins

A user interface (UI) component is composed entirely of attached movie clips. This means that all assets for a UI component can be external to the UI component movie clip, so they can be used by other components. For example, if your component needs check box functionality, you can reuse the existing CheckBox component assets.

The CheckBox component uses a separate movie clip to represent each of its states (FalseUp, FalseDown, Disabled, Selected, and so on). However, you can associate custom movie clips—called *skins*—with these states. At runtime, the old and new movie clips are exported in the SWF file. The old states simply become invisible to give way to the new movie clips. The ability to change skins during authoring and at runtime is called *skinning*.

To skin components, create a variable for every skin element (movie clip symbol) used in the component and set it to the symbol's linkage ID. This lets a developer set a different skin element just by changing a parameter in the component, as shown here:

```
var falseUpIcon = "mySkin";
```

The following example shows the skin variables for the various states of the CheckBox component:

```
var falseUpSkin:String = "";
var falseDownSkin:String = "";
var falseOverSkin:String = ""
var falseDisabledSkin:String = "";
var trueUpSkin:String = "";
var trueDownSkin:String = "";
var trueOverSkin:String = "";
var trueDisabledSkin:String = "";
var falseUpIcon:String = "CheckFalseUp";
var falseDownIcon:String = "CheckFalseDown";
var falseOverIcon:String = "CheckFalseOver";
var falseDisabledIcon:String = "CheckFalseDisabled";
var trueUpIcon:String = "CheckTrueUp";
var trueDownIcon:String = "CheckTrueDown";
var trueOverIcon:String = "CheckTrueOver";
var trueDisabledIcon:String = "CheckTrueDisabled";
```

About styles

You can use styles to register all the graphics in your component with a class and let that class control the color scheme of the graphics at runtime. No special code is necessary in the component implementations to support styles. Styles are implemented entirely in the base classes (UIObject and UIComponent) and skins.

To add a new style to a component, call `getStyle("styleName")` in the component class. If the style has been set on an instance, on a custom style sheet, or on the global style sheet, the value is retrieved. If not, you may need to install a default value for the style on the global style sheet.

For more information about styles, see "Using styles to customize component color and text" on page 82.

Registering skins to styles

The following example creates a component called Shape. This component displays a shape that is one of two skins: a circle or a square. The skins are registered to the `themeColor` style.

To register a skin to a style:

1. Create a new ActionScript file and copy the following code into it:

```
import mx.core.UIComponent;

class Shape extends UIComponent{

  static var symbolName:String = "Shape";
  static var symbolOwner:Object = Shape;
  var className:String = "Shape";

  var themeShape:String = "circle_skin"

  function Shape(){
  }

  function init(Void):Void{
    super.init();
  }

  function createChildren():Void{
    setSkin(1, themeShape);
    super.createChildren();
  }
}
```

2. Save the file as Shape.as.
3. Create a new Flash document and save it as Shape.fla in the same folder as Shape.as
4. Draw a circle on the Stage, select it, and press F8 to convert it to a movie clip.

   Give the circle the name and linkage identifier **circle_skin**.
5. Open the `circle_skin` movie clip and place the following ActionScript on Frame 1 to register the symbol with the style name `themeColor`:

```
mx.skins.ColoredSkinElement.setColorStyle(this, "themeColor");
```

6. Create a new movie clip for the component.

   Name the movie clip and linkage identifier **Shape**.
7. Create two layers. Place a `stop()` function in the first frame of the first layer. Place the symbol `circle_skin` in the second frame.

   This is the component movie clip. For more information, see "Creating a component movie clip" on page 138.

8. Open StandardComponents.fla as an external library, and drag the UIComponent movie clip to the Stage on the second frame of the Shape movie clip (with circle_skin).
9. Close StandardComponents.fla.
10. Select the Shape movie clip in the library, select Component Definition from the Library context menu (Windows: Right-click, Mac: control-click), and enter the AS 2.0 class name **Shape**.
11. Test the movie clip with the Shape component on the Stage.

    To change the theme color, set the style on the instance. The following code changes the color of a Shape component with the instance name `shape` to red:

    ```
    shape.setStyle("themeColor",0xff0000);
    ```

12. Draw a square on the Stage and convert it to a movie clip.

    Enter the linkage name **square_skin**, and make sure the Export in First Frame check box is selected.

NOTE	Because the movie clip isn't placed in the component, Export in First Frame must be selected so that the skin is available before initialization.

13. Open the `square_skin` movie clip and place the following ActionScript on Frame 1 to register the symbol with the style name `themeColor`:

    ```
    mx.skins.ColoredSkinElement.setColorStyle(this, "themeColor");
    ```

14. Place the following code on the instance of the Shape component on the Stage in the main Timeline:

    ```
    onClipEvent(initialize){
      themeShape = "square_skin";
    }
    ```

15. Test the movie clip with Shape on the Stage. The result should display a red square.

Registering a new style name

If you have created a new style name and it is a color style, add the new name to the `colorStyles` object in the StyleManager.as file (First Run\Classes\mx\styles\StyleManager.as). This example adds the `shapeColor` style:

```
// initialize set of inheriting color styles
  static var colorStyles:Object =
  {
    barColor: true,
    trackColor: true,
    borderColor: true,
    buttonColor: true,
    color: true,
    dateHeaderColor: true,
    dateRollOverColor: true,
    disabledColor: true,
    fillColor: true,
    highlightColor: true,
    scrollTrackColor: true,
    selectedDateColor: true,
    shadowColor: true,
    strokeColor: true,
    symbolBackgroundColor: true,
    symbolBackgroundDisabledColor: true,
    symbolBackgroundPressedColor: true,
    symbolColor: true,
    symbolDisabledColor: true,
    themeColor:true,
    todayIndicatorColor: true,
    shadowCapColor:true,
    borderCapColor:true,
    focusColor:true,
    shapeColor:true
  };
```

Register the new style name to the circle and square skins on Frame 1 of each skin movie clip, as follows:

```
mx.skins.ColoredSkinElement.setColorStyle(this, "shapeColor");
```

The color can be changed with the new style name by setting the style on the instance, as shown here:

```
shape.setStyle("shapeColor",0x00ff00);
```

Incorporating existing components within your component

In this section, you will build a simple LogIn component that incorporates Label, TextInput and Button components. This tutorial demonstrates how existing components are incorporated in new components by adding their uncompiled Flash (FLA) library symbols. The completed component files, LogIn.fla, LogIn.as and LogIn.swf are located in the examples folder on your hard disk:

- In Windows: the C:\Program Files\Macromedia\Flash 8\Samples and Tutorials\Samples\Components\Login folder.
- On the Macintosh: HD/Applications/Macromedia Flash 8/Samples and Tutorials/Samples/Components/Login folder.

The LogIn component provides an interface for entering a name and password. The API for LogIn has two properties, `name` and `password` for setting and getting the string values in the name and password TextInput fields. The LogIn component also dispatches a “click” event when the user clicks a button labeled LogIn.

- “Creating the LogIn Flash (FLA) file” on page 173
- “The LogIn class file” on page 176
- “Testing and exporting the LogIn component” on page 180

Creating the LogIn Flash (FLA) file

Start by creating a Flash (FLA) file that will hold our component symbol.

To create the LogIn FLA file:

1. In Flash, select File > New and create a new document.
2. Select File > Save As and save the file as LogIn.fla.
3. Create Select Insert > New Symbol. Name it **LogIn**, and select the Movie clip type radio button.

   If the Linkage section of the Create New Symbol dialog isn’t open, click the Advanced button to reveal it.
4. Select Export for ActionScript and deselect Export in First Frame.
5. Enter a linkage identifier.

   The default linkage identifier is LogIn. The rest of these steps assume you use the default value.

6. Enter **LogIn** in the AS 2.0 Class text box. This value is the component class name.

   If you put the class in a package, enter the entire package name. For example, mx.controls.CheckBox denotes the CheckBox class in the mx.controls package.

7. Click OK.

   Flash opens in symbol-editing mode.

8. Insert a new layer. Name the top layer **Actions** and the bottom layer **Assets**.

9. Select Frame 2 in the Assets layer and insert a keyframe (F6).

   This is the structure of the component movie clip: an Actions layer and an Assets layer. The Actions layer has 1 keyframe and the Assets layer has 2 keyframes.

10. Select Frame 1 in the Actions layer and open the Actions panel (F9). Enter a `stop()` global function.

    This prevents the movie clip from proceeding to Frame 2.

11. Select File > Import > Open External Library and select the StandardComponents.fla file from the Configuration/ComponentFLA folder.

    - In Windows: \Program Files\Macromedia\Flash 8*language*\Configuration\ComponentFLA\StandardComponents.fla.
    - On the Macintosh: HD/Applications/Macromedia Flash 8/Configuration/ComponentFLA/StandardComponents.fla

    NOTE For information about folder locations, see "Configuration folders installed with Flash" in Using Flash.

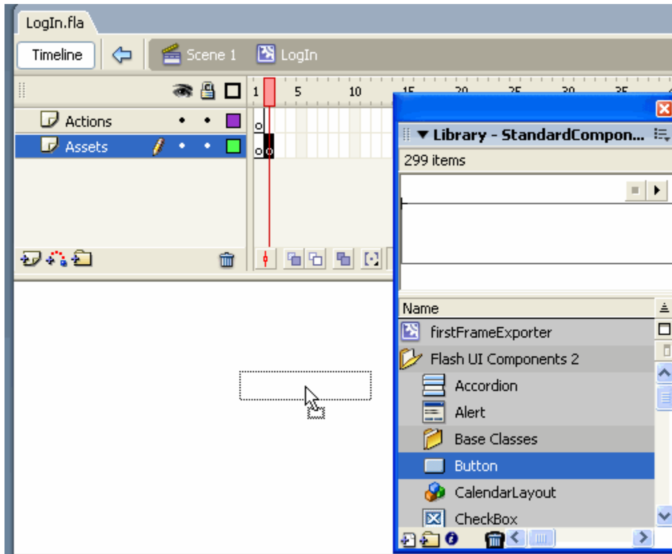
12. Select Frame 2 in the Assets layer. From within the StandardComponents.fla library, browse to the Flash UI Components 2 folder. Drag a Button, Label and TextInput component symbol to Frame 2 of the Assets layer.

    Asset dependencies for these components are automatically copied to your LogIn.fla library.

    All the component assets are added to Frame 2 of the Assets layer. Because there is a `stop()` global function on Frame 1 of the Actions layer, the assets in Frame 2 will not be seen as they are arranged on the Stage.

    You add assets to Frame 2 for two reasons:

- So that all assets are automatically copied into the library and are available to instantiate dynamically and access their methods, properties, and events.
- Placing assets in a frame ensures they are loaded more smoothly as the movie is streamed, so you do not need to set the assets in the library to be exported before the first frame. This method prevents an initial data transfer spike that could cause download delays or long pauses.



Dragging a Button component symbol from the library in StandardComponents.fla to Frame 2 of the Assets layer of LogIn.fla

13. Close the StandardComponents.fla library.
14. In the Assets layer, select Frame 1. Drag the BoundingBox movie clip from the LogIn.fla library (inside the Component Assets folder) to the Stage.
15. Name the BoundingBox instance **boundingBox_mc**.

16. Use the Info panel to resize the BoundingBox to the size of the LogInFinal movie clip (340, 150), and position it at 0, 0.

    The BoundingBox instance is used to create the component's live preview and allow the user to handle resize the component during authoring. You must set the size of the bounding box so that it can enclose all the graphical elements in your component.

    > **NOTE**
    > If you are extending a component (including any version 2 component) you must keep instance names already in use by that component as its code will refer to those instance names. For example, if you include a version 2 component that is already using the instance name boundingBox_mc, do not rename it. For your own components, you can choose any instance name that is unique and that does not conflict with an existing name within the same scope.

17. Select the LogIn movie clip in the library, and select Component Definition from the Library context menu (Windows: Right-click, Mac: control-click).
18. In the AS 2.0 Class text box, enter **LogIn**.

    This value is the name of the ActionScript class. If the class is in a package, the value is the full package. For example, mx.controls.CheckBox denotes the CheckBox class in the mx.controls package.

19. Click OK.
20. Save the file.

The LogIn class file

The following code is the ActionScript class for the LogIn component. Please read the comments in the code for a description of each section. (For detailed information on the elements of a component class file, see "Overview of a component class file" on page 144).

To create this file, you can create a new ActionScript file in Flash, or use any other text editor. Save the file as LogIn.as in the same folder as the LogIn.fla file.

You can copy or type the following LogIn component ActionScript class code into your new LogIn.as file. Typing rather than copying the code will help you become familiar with each element of the component code.

```
/* Import the packages so they can be referenced
   from this class directly. */
import mx.core.UIComponent;
import mx.controls.Label;
import mx.controls.TextInput;
import mx.controls.Button;

// Event metadata tag
[Event("change")]
```

```
[Event("click")]
class LogIn extends UIComponent
{
  /* Components must declare these member variables to be proper
     components in the components framework. */
  static var symbolName:String = "LogIn";
  static var symbolOwner:Object = LogIn;
  var className:String = "LogIn";

  // The component's graphical representation.
  private var name_label:MovieClip;
  private var password_label:MovieClip;
  private var name_ti:MovieClip;
  private var password_ti:MovieClip;
  private var login_btn:MovieClip;
  private var boundingBox_mc:MovieClip;
  private var startDepth:Number = 10;

  /* Private member variables available publicly through getter/setters.
     These represent the name and password InputText string values. */
  private var __name:String;
  private var __password:String;

  /* Constructor:
     While required for all classes, v2 components require
     the contstructor to be empty with zero arguments.
     All initialization takes place in a required init
     method after the class instance has been constructed. */
  function LogIn() {
  }

  /* Initialization code:
     The init method is required for v2 components. It must also
     in turn call its parent class init() method with super.init().
     The init method is required for components extending UIComponent. */
  function init():Void {
    super.init();
    boundingBox_mc._visible = false;
    boundingBox_mc._width = 0;
    boundingBox_mc._height = 0;
  }

  /* Create child objects needed at start up:
     The createChildren method is required for components
     extending UIComponent. */
  public function createChildren():Void {
    name_label = createObject("Label", "name_label", this.startDepth++);
    name_label.text = "Name:";
    name_label._width = 200;
    name_label._x = 20;
```

```
  name_label._y = 10;

  name_ti = createObject("TextInput", "name_ti",
this.startDepth++,{_width:200,_heigh:22,_x:20,_y:30});
  name_ti.html = false;
  name_ti.text = __name;
  name_ti.tabIndex = 1;
  /* Set this text input field to have focus.
     Note: Make sure to set select Control > Disable Keyboard Shortcuts
     in the Flash Debugger if it is not already selected, otherwise
     the focus may not set when testing. */
  name_ti.setFocus();

  name_label = createObject("Label", "password_label",
this.startDepth++,{_width:200,_heigh:22,_x:20,_y:60});
  name_label.text = "Password:";

  password_ti = createObject("TextInput", "password_ti",
this.startDepth++,{_width:200,_heigh:22,_x:20,_y:80});
  password_ti.html = false;
  password_ti.text = __password;
  password_ti.password = true;
  password_ti.tabIndex = 2;

  login_btn = createObject("Button", "login_btn",
this.startDepth++,{_width:80,_heigh:22,_x:240,_y:80});
  login_btn.label = "LogIn";
  login_btn.tabIndex = 3;
  login_btn.addEventListener("click", this);

  size();

}

/* The draw method is required for v2 components.
   It is invoked after the component has been
   invalidated by someone calling invalidate().
   This batch's up the changes into one redraw, rather
   than doing them all individually. This approach leads
   to more efficiency and better centralization of code. */
function draw():Void {
  super.draw();
}

/* The size method is invoked when the component's size
   changes. This is an opportunity to resize the children,
   The size method is required for components extending UIComponent. */
function size():Void {
  super.size();
  // Cause a redraw in case it is needed.
```

```
    invalidate();
  }

  /* Event Handler:
     Called by the LogIn button when it receives a mouse click.
     Since we want this event to be accessible outside of the scope of
     this component, The click event is dispatched using dispatchEvent. */
  public function click(evt){
    // Update the member variables with the input field contents.
    __name = name_ti.text;
    __password = password_ti.text;
    // Dispatch a click event when the button fires one.
    dispatchEvent({type:"click"});
  }

  /* This is the getter/setter for the name property.
     The [Inspectable] metadata makes the property appear
     in the Property inspector and allows a default value
     to be set. By using a getter/setter you can call invalidate
     and force the component to redraw when the value is changed. */
  [Bindable]
  [ChangeEvent("change")]
  [Inspectable(defaultValue="")]
  function set name(val:String){
    __name = val;
    invalidate();
  }

  function get name():String{
    return(__name);
  }

  [Bindable]
  [ChangeEvent("change")]
  [Inspectable(defaultValue="")]
  function set password(val:String){
    __password=val;
    invalidate();
  }

  function get password():String{
    return(__password);
  }

}
```

Testing and exporting the LogIn component

You've created the Flash file that contains the graphical elements, the base classes and the class file that contains all the functionality of the LogIn component. Now it's time to test the component.

Ideally, you would test the component as you work, especially while you're writing the class file. The fastest way to test as you work is to convert the component to a compiled clip and use it in the component's FLA file.

When you're completely finished creating a component, export it as a SWC file. For more information, see "Exporting and distributing a component" on page 182.

To test the LogIn component:

1. In the LogIn.fla file, select the LogIn movie clip in the library and select Convert to Compiled Clip from the Library context menu (Windows: Right-click, Mac: control-click).

   A compiled clip is added to the library with the name LogIn SWF. You are compiling the movie clip to test it, only. Otherwise, you would follow the instructions later in this section to export the LogIn movie clip.

   > NOTE: If you've already created a compiled clip (for example, if this is the second or third time you're testing), a Resolve Library Conflict dialog box appears. Select Replace Existing Items to add the new version to the document.

2. Drag LogIn SWF to the Stage in frame 1 of the main Timeline (make sure you're in the main Timeline, Scene 1, not the movie clip timeline).

   You can set the name and password property in the Parameters tab or the Component Inspector. This is useful if you want default text such as "Enter your name here" to appear before the user has entered anything. When you set its name and/or password property, the default text in the name and password InputText sub-components will change accordingly at runtime.

   To test the `value` property at runtime, name the LogIn instance on the Stage **myLogin** and add the following code to Frame 1 in the main Timeline:

```
// Creates a text field in which to view the login values.
createTextField("myLoginValues",10,10,10,340,40)
myLoginValues.border = true;
// Event handler for the login component instance's dispatched click
  event.
function click(evt){
/* Here is where authentication would occur.
```

```
    For example the name and password would be passed to a web service
    which authenticates the name and password and returns a session ID
    and/or permission roles attributed to the user. */
    myLoginValues.text = "Processing...\r";
    myLoginValues.text += "Name: " + myLogin.name + "  Password: " + 
    myLogin.password;
}

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

3. Select Control > Test Movie to test the component in Flash Player.

> **NOTE** Since you are testing this component within your original document, you may see a warning message about having the same linkage identifier for two symbols. The component will still work. In practice, you will use the new component within another document in which case the linkaged identifier should be unique.

To export the LogIn component:

1. In the LogIn.fla file, select the LogIn movie clip in the library and select Component Definition from the Library context menu (Windows: Right-click, Mac: control-click).
2. Check the Display in Components panel.
3. Click OK.
4. In the LogIn.fla file, select the LogIn movie clip in the library, again, and select Export SWC File from the Library context menu (Windows: Right-click, Mac: control-click).
5. Select a location to save the SWC file.

   If you save it to the Components folder in the user-level configuration folder, you can reload the Components panel without restarting Flash and the component appears in the Components panel.

> **NOTE** For information about folder locations, see "Configuration folders installed with Flash" in *Getting Started with Flash*.

The completed LogIn component

Exporting and distributing a component

Flash exports components as component packages (SWC files). Components may be distributed as SWC files or as FLA files. (See the article on Macromedia DevNet at www.macromedia.com/support/flash/applications/creating_comps/creating_comps12.html for information about distributing a component as a FLA.)

The best way to distribute a component is to export is as a SWC file, because SWC files contain all the ActionScript, SWF files, and other optional files needed to use the component. SWC files are also useful if you are working at the same time on a component and the application that uses the component.

SWC files can be used to distribute components for use in Macromedia Flash 8, Macromedia Dreamweaver MX 2004, and Macromedia Director MX 2004.

Whether you're developing a component for someone else's use, or for your own, it's important to test the SWC file as an ongoing part of component development. For example, problems can arise in a component's SWC file that don't appear in the FLA file.

This section describes a SWC file and explains how to import and export SWC files in Flash.

Understanding SWC files

A SWC file is a zip-like file (packaged and expanded by means of the PKZIP archive format) generated by the Flash authoring tool.

The following table describes the contents of a SWC file:

File	Description
catalog.xml	(Required) Lists the contents of the component package and its individual components, and serves as a directory to the other files in the SWC file.
ActionScript (AS) files	If you create the component with Flash Professional 8, the source code is one or more ActionScript files that contain a class declaration for the component. The compiler uses the source code for type checking when a component is extended. The AS file is not compiled by the authoring tool because the compiled bytecode is already in the implementing SWF file. The source code may contain intrinsic class definitions that contain no function bodies and are provided purely for type checking.
SWF files	(Required) SWF files that implement the components. One or more components can be defined in a single SWF file. If the component is created with Flash 8, only one component is exported per SWF file.

File	Description
Live Preview SWF files	(Optional) If specified, these SWF files are used for live preview in the authoring tool. If omitted, the SWF files that implement the component are used for live preview instead. The Live Preview SWF file can be omitted in nearly all cases; it should be included only if the component's appearance depends on dynamic data (for example, a text field that shows the result of a web service call).
SWD file	(Optional) A SWD file corresponding to the implementing SWF file that allows you to debug the SWF file. The filename is always the same as that of the SWF file, but with the extension.swd.
PNG file	(Optional) A PNG file containing the 18 x 18, 8-bit-per-pixel icon that you use to display a component icon in the authoring tool user interfaces. If no icon is supplied, a default icon is displayed. (See "Adding an icon" on page 185.)
Property inspector SWF file	(Optional) A SWF file that you use as a custom Property inspector in the authoring tool. If you omit this file, the default Property inspector is displayed to the user.

You can optionally include other files in the SWC file, after you generate it from the Flash environment. For example, you might want to include a Read Me file, or the FLA file if you want users to have access to the component's source code. To add additional files, use the Macromedia Extension Manager (see www.macromedia.com/exchange/em_download/).

SWC files are expanded into a single directory, therefore each component must have a unique file name to prevent conflicts.

Exporting SWC files

Flash provides the ability to export SWC files by exporting a movie clip as a SWC file. When exporting a SWC file, Flash reports compile-time errors as if you were testing a Flash application.

There are two reasons to export a SWC file:

- To distribute a finished component
- To test during development

Exporting a SWC for a completed component

You can export components as SWC files that contain all the ActionScript, SWF files, and other optional files needed to use the component.

To export a SWC file for a completed component:

1. Select the component movie clip in the Flash library.
2. Right-click (Windows) or control-click (Mac) to open the Library context menu.
3. Select Export SWC File from the Library context menu.
4. Save the SWC file.

Testing a SWC during development

At different stages of development, it's a good idea to export the component as a SWC and test it in an application. If you export the SWC to the Components folder in your user-level Configuration folder, you can reload the Components panel without quitting and restarting Flash.

To test a SWC during development:

1. Select the component movie clip in the Flash library.
2. Right-click (Windows) or control-click (Mac) to open the Library context menu.
3. Select Export SWC File from the Library context menu.
4. Browse to the Components folder in your user-level configuration folder.

   Configuration/Components

   > NOTE: For information about the location of the folder, see "Configuration folders installed with Flash" in *Getting Started with Flash*.

5. Save the SWC file.
6. Select Reload from the Components panel's options menu.

   The component appears in the Component panel.
7. Drag the component from the Component panel into a document.

Importing component SWC files into Flash

When you distribute your components to other developers, you can include the following instructions so that they can install and use them immediately.

To import a SWC file:

1. Copy the SWC file into the Configuration/Components directory.
2. Restart Flash.

   The component's icon should appear in the Components panel.

Final steps in component development

After you create the component and prepare it for packaging, you can add an icon and a tool tip. To make sure you completed all the necessary steps, you can also refer to the "Component development checklist" on page 186.

Adding an icon

You can add an icon that represents your component in the Components panel of the Flash authoring environment.

To add an icon for your component:

1. Create a new image.

   The image must measure 18 pixels square, and you must save it in PNG format. It must be 8-bit with alpha transparency, and the upper left pixel must be transparent to support masking.

2. Add the following definition to your component's ActionScript class file before the class definition:

   ```
   [IconFile("component_name.png")]
   ```

3. Add the image to the same directory as the FLA file. When you export the SWC file, Flash includes the image at the root level of the archive.

Adding a tooltip

Tooltips appear when a user rolls the mouse over your component name or icon in the Components panel of the Flash authoring environment.

You define a tooltip in the Component Definition dialog box. You can access this dialog box from the Library options menu (Windows: Right-click, Mac: control-click) of the component's FLA file.

To add a tooltip in the Component Definition dialog box:

1. With the FLA file of your component open in Flash, make sure the Library is visible (Window > Library menu).

2. Click the Library options menu (Windows: Right-click, Mac: Control-click).

   The Library options menu is on the right side of the Library title bar, and appears as an icon of three lines and a down triangle.

3. Select the Component Definition option.

4. In the Component Definition dialog box, under Options, select Display in the Components Panel.

   The Tool tip text box becomes editable.
5. Enter the tooltip text for your component in the Tool tip text box.
6. Click OK to save the changes.

Component development checklist

When you design a component, use the following practices:

- Keep the file size as small as possible.
- Make your component as reusable as possible by generalizing functionality.
- Use the RectBorder class (mx.skins.halo.RectBorder) rather than graphical elements to draw borders around objects. (See “RectBorder class” in the *Components Language Reference*.)
- Use tag-based skinning.
- Define the `symbolName`, `symbolOwner`, and `className` variables.
- Assume an initial state. Because style properties are now on the object, you can set initial settings for styles and properties so your initialization code does not have to set them when the object is constructed, unless the user overrides the default state.
- When defining the symbol, do not select the Export in First Frame option unless absolutely necessary. Flash loads the component just before it is used in your Flash application, so if you select this option, Flash preloads the component in the first frame of its parent. The reason you typically do not preload the component in the first frame is for considerations on the web: the component loads before your preloader begins, defeating the purpose of the preloader.
- Avoid multiple frame movie clips (except for the two-frame Assets layer).
- Always implement `init()` and `size()` methods and call `Super.init()` and `Super.size()` respectively, but otherwise keep them lightweight.
- Avoid absolute references, such as `_root.myVariable`.
- Use `createClassObject()` instead of `attachMovie()`.
- Use `invalidate()` and `invalidateStyle()` to invoke the `draw()` method instead of calling `draw()` explicitly.
- When incorporating Flash components into your component, use their uncompiled movie symbols located in the library of the StandardComponents.fla file from the Configuration/ComponentFLA folder.

CHAPTER 7

Collection Properties

7

When you create a new custom component in Macromedia Flash, you can make property values available for editing by the user. These properties are called *collection properties*. The property values can be edited by the user in the Values dialog box (opened from a text box within the Parameters tab for your component).

Components usually include functionality for a specific task, while remaining flexible for a range of requirements by the component user. For a component to be flexible, the properties within the component need to be flexible (in other words, for some components, the properties can be changed by the component user, as well as by the property values).

Collection properties enable you to create an indeterminate number of editable properties in an object model. Flash provides a Collection class to help you manage those properties through the Component inspector.

Specifically, the Collection class is a helper class used to manage a group of related objects, each called a *collection item*. If you define a property of your component as a collection item and make it available to users through the Component inspector, they can add, delete, and modify collection items in the Values dialog box while authoring.

You define collections and collection items as follows:

- Define a collection property using the Collection metadata tag in a component's ActionScript file. For more information, see "About the Collection tag" on page 157.
- Define a collection item as a class in a separate ActionScript file that contains its own inspectable properties.

In Flash, Collections make it easier for you to manage groups of related items programmatically. (In previous versions of Flash, component authors managed groups of related items through multiple programmatically synchronized arrays).

In addition to the Values dialog box, Flash provides the Collection and Iterator interfaces to manage Collection instances and values programmatically. See "Collection interface (Flash Professional only)" and "Iterator interface (Flash Professional only)" in the *Components Language Reference*.

This chapter contains the following sections:

Defining a collection property 188
Simple collection example 189
Defining the class for a collection item 191
Accessing collection information programmatically. 191
Exporting components that have collections to SWC files. 194
Using a component that has a collection property 194

Defining a collection property

You define a collection property by using the Collection tag in a component's ActionScript file. For more information, see "About the Collection tag" on page 157.

NOTE	This section assumes that you know how to create components and inspectable component properties.

To define a collection property:

1. Create a FLA file for your component. See "Creating a component movie clip" on page 138.
2. Create an ActionScript class. See "Creating the ActionScript class file" on page 143.
3. In the ActionScript class, insert a Collection metadata tag. For more information, see "About the Collection tag" on page 157.
4. Define `get` and `set` methods for the collection in the component's ActionScript file.

5. Add the utilities classes to your FLA file by selecting Window > Common Libraries > Classes and dragging UtilsClasses into the component's library.

   UtilsClasses contains the mx.utils.* package for the Collection interface.

   > **NOTE** Because UtilsClasses is associated with the FLA file, not the ActionScript class, Flash throws compiler errors when you check syntax while viewing the component's ActionScript class.

6. Code a class that contains the collection item properties.

   See "Defining the class for a collection item" on page 191.

Simple collection example

The following is a simple example of a component class file called MyShelf.as. This example contains a collection property along with a minimal set of imports, methods, and declarations for a component that inherits from the UIObject class.

If you import mx.utils.* in this example, the class names from mx.utils no longer need to be fully qualified. For instance, mx.utils.Collection can be written as Collection.

```
import mx.utils.*;
// standard class declaration
class MyShelf extends mx.core.UIObject
{
// required variables for all classes
  static var symbolName:String = "MyShelf";
  static var symbolOwner:Object = Object(MyShelf);
  var className:String = "MyShelf";

// the Collection metadata tag and attributes
   [Collection(variable="myCompactDiscs",name="My Compact
  Discs",collectionClass="mx.utils.CollectionImpl",
  collectionItem="CompactDisc", identifier="Title")]

// get and set methods for the collection
  public function get MyCompactDiscs():mx.utils.Collection
  {
    return myCompactDiscs;
  }
  public function set MyCompactDiscs(myCDs:mx.utils.Collection):Void
  {
    myCompactDiscs = myCDs;
  }

  // private class member
  private var myCompactDiscs:mx.utils.Collection;
```

```
// You must code a reference to the collection item class
// to force the compiler to include it as a dependency
// within the SWC
  private var collItem:CompactDisc;

// You must code a reference to the mx.utils.CollectionImpl class
// to force the compiler to include it as a dependency
// within the SWC
  private var coll:mx.utils.CollectionImpl;

  // required methods for all classes
  function init(Void):Void {
    super.init();
  }
  function size(Void):Void {
    super.size();
  }
}
```

To create a FLA file to accompany this class for testing purposes:

1. In Flash, select File > New and create a Flash document.
2. Select Insert > New Symbol. Give it the name, linkage identifier, and AS 2.0 class name **MyShelf**.
3. Deselect Export in First Frame and click OK.
4. Select the MyShelf symbol in the library and choose Component Definition from the Library options menu. Enter the ActionScript 2.0 class name **MyShelf**.
5. Select Window > Common Libraries > Classes, and drag UtilClasses to the library of MyShelf.fla.
6. In the MyShelf symbol's Timeline, name one layer **Assets**. Create another layer and name it **Actions**.
7. Place a `stop()` function on Frame 1 in the Actions layer.
8. Select Frame 2 in the Assets layer and select Insert > Timeline > Keyframe.
9. Open StandardComponents.fla from the Configuration/ComponentFLA folder, and drag an instance of UIObject to the Stage of MyShelf in Frame 2 of the Assets layer.

   You must include UIObject in the component's FLA file because, as you can see in the above class file, MyShelf extends UIObject.
10. In Frame 1 of the Assets layer, draw a shelf.

    This can be a simple rectangle; it's just a visual representation of the MyShelf component to use for learning purposes.
11. Select the MyShelf movie clip in the library, and select Convert to Compiled Clip.

This allows you to drag the MyShelf SWF file (the compiled clip that's added to the library) into the MyShelf.fla file to test the component. Whenever you recompile the component, a Resolve Library Conflict dialog box appears, because an older version of the component already exists in the library. Choose to replace existing items.

> **NOTE** You should have already created the CompactDisc class; otherwise, you'll get compiler errors when converting to a compiled clip.

Defining the class for a collection item

You code the properties for a collection item in a separate ActionScript class, which you define as follows:

- Define the class such that it does not extend UIObject or UIComponent.
- Define all properties using the Inspectable tag.
- Define all properties as variables. Do not use `get` and `set` (getter/setter) methods.

The following is a simple example of a collection item class file called CompactDisc.as.

```
class CompactDisc{
   [Inspectable(type="String", defaultValue="Title")]
   var title:String;
   [Inspectable(type="String", defaultValue="Artist")]
   var artist:String;
}
```

To view the CompactDisc.as class file, see "Simple collection example" on page 189.

Accessing collection information programmatically

Flash provides programmatic access to collection data through the Collection and Iterator interfaces. The Collection interface lets you add, modify, and remove items in a collection. The Iterator interface allows you to loop through the items in a collection.

There are two scenarios in which to use the Collection and Iterator interfaces:

- "Accessing collection information in a component class (AS) file" on page 192
- "Accessing collection items at runtime in a Flash application" on page 193

Advanced developers can also create, populate, access, and delete collections programmatically; for more information, see "Collection interface (Flash Professional only)" in the *Components Language Reference*.

Accessing collection information in a component class (AS) file

In a component's class file, you can write code that interacts with collection items defined during authoring or at runtime.

To access collection item information in a component class file, you can use any of the following approaches.

- The Collection tag includes a `variable` attribute, for which you specify a variable of type mx.utils.Collection. Use this variable to access the collection, as shown in this example:

  ```
  [Collection(name="LinkButtons", variable="__linkButtons",
    collectionClass="mx.utils.CollectionImpl", collectionItem="ButtonC",
    identifier="ButtonLabel")]
  public var __linkButtons:mx.utils.Collection;
  ```

- Access the Iterator interface for the collection items by calling the `Collection.getIterator()` method, as shown in this example:

  ```
  var itr:mx.utils.Iterator = __linkButtons.getIterator();
  ```

- Use the Iterator interface to step through the items in the collection. The `Iterator.next()` method returns an Object, so you must define the type of your collection item, as shown in this example:

  ```
  while (itr.hasNext()) {
    var button:ButtonC = ButtonC(itr.next());
    ...
  }
  ```

- Access collection item properties, as appropriate for your application, as shown in this example:

  ```
  item.label =  button.ButtonLabel;

  if (button.ButtonLink != undefined) {
    item.data = button.ButtonLink;
  }
  else {
    item.enabled = false;
  }
  ```

Accessing collection items at runtime in a Flash application

If a Flash application uses a component that has a collection property, you can access the collection property at runtime. This example adds several items to a collection property using the Values dialog box and displays them at runtime using the Collection and Iterator APIs.

To access collection items at runtime:

1. Open the MyShelf.fla file that you created earlier.

   See "Simple collection example" on page 189.

   This example builds on the MyShelf component and CompactDisc collection.

2. Open the Library panel, drag the component onto the Stage, and give it an instance name.

   This example uses the instance name myShelf.

3. Select the component, open the Component inspector, and display the Parameters tab. Click the line that contains the collection property, and click the magnifying glass to the right of the line. Flash displays the Values dialog box.

4. Use the Values dialog box to enter values into the collection property.

5. With the component selected on the Stage, open the Actions panel and enter the following code (which must be attached to the component):

```
onClipEvent (mouseDown) {
  import mx.utils.Collection;
  import mx.utils.Iterator;
  var myColl:mx.utils.Collection;
  myColl = _parent.myShelf.MyCompactDiscs;

  var itr:mx.utils.Iterator = myColl.getIterator();
  while (itr.hasNext()) {
    var cd:CompactDisc = CompactDisc(itr.next());
    var title:String = cd.Title;
    var artist:String = cd.Artist;
    trace("Title: " + title + " Artist: " + artist);
  }
}
```

   To access a collection, use the syntax `componentName.collectionVariable`; to access an iterator and step through the collection items, use `componentName.collectionVariable.getIterator()`.

6. Select Control > Test Movie and click the shelf to see the collection data in the Output panel.

Exporting components that have collections to SWC files

When you distribute a component that has a collection, the SWC file must contain the following dependent files:

- Collection interface
- Collection implementation class
- Collection item class
- Iterator interface

Of these files, your code typically uses the Collection and Iterator interfaces, which marks them as dependent classes. Flash automatically includes dependent files in the SWC file and output SWF file.

However, the collection implementation class (mx.utils.CollectionImpl) and component-specific collection item class are not automatically included in the SWC file.

To include the collection implementation class and collection item class in the SWC file, you define private variables in your component's ActionScript file, as the following example shows:

```
// collection item class
private var collItem:CompactDisc;
// collection implementation class
private var coll:mx.utils.CollectionImpl;
```

For more information on SWC files, see "Understanding SWC files" on page 182.

Using a component that has a collection property

When you use a component that includes a collection property, you typically use the Values dialog box to establish the items in the collection.

To use a component that includes a collection property:

1. Add the component to the Stage.
2. Use the Property inspector to name the component instance.
3. Open the Component inspector and display the Parameters tab.
4. Click the line that contains the collection property, and click the magnifying glass to the right of the line.

   Flash displays the Values dialog box.

5. Click the Add (+) button and define a collection item.
6. Click the Add (+), Delete (-), and arrow buttons to add, modify, move, and delete collection items.
7. Click OK.

For information on accessing a collection programmatically, see "Accessing collection items at runtime in a Flash application" on page 193.

Index

Numerics

9-slice not supported 19

A

accessibility 20
ActionScript class files 143
audience for this document 8

B

best practices for component development 186
Binding tab, in sample application (tutorial) 30
bitmap caching not supported 19
broadcaster 64

C

class file
  about 144
  example 143, 189
class keyword 146
class style sheets 82
classes
  and component inheritance 18
  creating references to (tutorial) 25
  creating. *See* creating components
  defining 146
  extending 137
  importing 146
  selecting a parent class 136
  UIComponent 137
  UIObject 137
className variable 147
code hints, triggering 58
collection item 191
collection properties
  accessing programmatically 191
  defining 188
  defining classes 191
  example 189
  exporting components 194
  using 194
Collection tag 157
colors
  customizing 82
  setting style properties 92
columns
  adding (tutorial) 31
compiled clips
  about 19
  in Library panel 54
component class file. *See* class file
Component inspector
  Binding tab 30
  setting parameters 55
component parameters
  about 55
  defining 159
  inspectable 151
  setting 55
  viewing 55
  See also individual component names
components 143
  ActionScript class 143
  adding an icon 185
  adding at runtime 52
  adding to Flash documents 50
  adding tooltips 185
  architecture 17
  assigning skins 168
  available in Flash MX editions 12
  categories, described 16

class file example 143
class overview 144
className variable 147
common events 167
creating movie clips 138
creating subobjects 162
defining draw() method 165
defining init() method 162
defining parameters 159
defining size() method 165
defining variables 148
deleting 57
development checklist 186
dispatching events 166
editing movie clips 139
events 63
example of building a component 127, 173
example of class file with collection 189
exporting and distributing 182
exporting component as SWC 183
exporting SWC files 183
extending classes 137
Flash Player support 17
getter/setter methods 148
importing SWC files 184
inheritance 18
installing 12
invalidation, about 166
loading 62
metadata tags 149
metadata, ComponentTask tag 158
preloading 60
previewing 60
registering skins to styles 169
selecting a parent class 136
selecting symbol names 147
source files 125
structure of 126
styles 169
symbolOwner variable 147
testing SWC files 184
using in an application (tutorial) 21
See also individual component names
Components panel 50
ComponentTask tag
JavaScript (JSFL) 158
createClassObject() method 162
CSSStyleDeclaration 86, 87
customizing
text 82
customizing color and text, using style sheets 82

D

data binding, with XML file (tutorial) 29
data grids. *See* DataGrid component
data types, setting for instances (tutorial) 27
DataGrid component
adding columns (tutorial) 31
binding to DataSet (tutorial) 29
DataSet component, binding to XMLConnector and DataGrid (tutorial) 29
defaultPushButton property 59
Delegate class (tutorial) 73
deleting components 57
DepthManager class, overview 59
Dial component 127, 173
dispatcher (event broadcaster) 64
dispatching events 166
documentation
guide to terminology 9
overview 8
draw() method, defining 165

E

event listeners. *See* listeners
Event metadata tag 153
events
about 63
common 167
delegating scope 73
dispatching 166
event object 77
handler functions 63
metadata 153
See also individual component names
exporting components 182
extending classes 137

F

Flash JavaScript (JSFL), ComponentTask tag 158
Flash MX editions and available components 12
Flash Player
and components 17
support 62
FlashType not supported 19
FocusManager class, creating focus navigation for 58

G

getter/setter methods 148
global style declarations 86
grids. *See* DataGrid component

H

Halo theme 108
handleEvent callback function 68
handler functions 63

I

icon, for a component 185
import statement 146
inheritance, in version 2 components 18
init() method, defining 162
inspectable parameters 151
installing components 12
instances
  setting styles on 84
  style declarations 82
invalidate() method 166

L

Label component
  tutorial 40
library
  compiled clips in 54
  Library panel 54
  StandardComponents 139
linkage identifiers for skins 96
listeners
  about 64
  functions 69
  objects 65
  scope 71
  using with component instances (tutorial) 37
  using with components (tutorial) 32
Live Preview feature 60
loading, components 62

M

metadata
  about 149
  Collection tag 157
  ComponentTask tag 158
  Event tag 153
  Inspectable tag 151
  tags, list of 150
movie clips
  creating 138
  defining as component 141
MovieClip class, extending 138

O

on() event handler 78

P

packages 18
parameters. *See* component parameters
parent class, selecting 136
preloading components 60
previewing components 60
Property inspector 55
prototype 106

R

resources, additional Macromedia 9

S

Sample theme 108
screen readers, accessibility 20
ScrollPane component
  tutorial 40
size() method, defining 165
skin properties
  changing in the prototype 106
  setting 96
skinning components 96
skins
  applying to components 101
  applying to subcomponents 103
  creating variables for 168
  editing 99
  linkage identifiers 96
  See also individual component names

StandardComponents library 139
style declarations
  custom 87
  default class 89
  global 86
  setting class 89
style properties, color 92
style sheets
  class 82
  custom 82
styles
  about 82
  creating components 169
  determining precedence 92
  setting 82
  setting custom 87
  setting global 86
  setting on instance 84
  using (tutorial) 27
  See also individual component names
subclasses, using to replace skins 106
subcomponents, applying skins 103
subobjects, creating 162
superclass keyword 146
SWC files
  about 19
  and compiled clips 19
  exporting 183
  exporting collection properties 194
  file format 182
  importing 184
  testing 184
symbolName variable 147
symbolOwner variable 147
system requirements for components 8

T

tabbing 58
tags. *See* metadata
terminology in documentation 9
testing SWC files 184
text, customizing 82
TextInput component (tutorial) 40
themes
  about 108
  applying 113
  creating 111
tip text, adding 185
typographical conventions 9

U

UIComponent class
  and component inheritance 18
  overview 137
UIObject class
  about 137
upgrading version 1 components 62

V

Values dialog box 187
variables, defining 148
version 1 components, upgrading 62
version 2 components
  and Flash Player 17
  benefits 16
  inheritance 18

W

web service, connecting to (tutorial) 28
WebService class (tutorial) 28

X

XMLConnector component
  binding to DataSet component (tutorial) 29
  loading an external XML file (tutorial) 31
  specifying schema (tutorial) 29