Adobe

# Best practices for accessibility with Adobe® Flex® 4

## Building applications that can be used by individuals with disabilities

**Table of contents**


1: Achieving accessibility
2: An overview of assistive technologies
2: Accessibility testing overview
3: Enabling accessibility
4: Embedding accessible Flex applications
5: Accessible Flex components
6: Accessibility properties
7: Setting component name information
9: Checking for the presence of assistive technology
10: Textual equivalents
15: Keyboard accessibility
23: Reading order
29: Reading and tab order in practice
31: Skipping repetitive components
32: Forms
39: Focus
44: Color
46: Flickering
47: Resources


To reach as wide an audience as possible, developers of Flex applications need to ensure that their applications built with Adobe Flex can be used by individuals with disabilities. Conforming to regulatory requirements such as Section 508 of the Rehabilitation Act or the Web Content Accessibility Guidelines (WCAG) 2.0 helps developers build applications that are accessible to users, regardless of disability. Compliance with such guidelines may be a formal requirement on some projects.

This document covers general principles and best practices of accessible design that apply to all Flex applications independent of the disabilities the target user audience may have. Developers who are familiar with accessible software design practices used in other development contexts will likely be familiar with the best practices described in this document for Flex 4 applications.

## Achieving accessibility

An application is not made accessible by simply switching on or enabling accessibility. Likewise, accessible applications are not created merely by exposing accessibility information for individual components, although this is a necessary first step. Accessibility is achieved through correct design of the application and user interaction, and by applying accessibility information to individual components, groups of components, and to the application as a whole. Developers must consider how user interface items interact together, how they flow from one to another, and how the user will interact with them. This document includes guidelines for the design and development of accessible Flex 4 applications, and covers the following topics:

- Enabling Accessibility in Flex projects
- Embedding Accessible Flex Applications
- Accessible Flex Components
- Accessibility Properties
- Setting Component Name Information
- Checking for the Presence of Assistive Technology
- Textual Equivalents
- Keyboard Accessibility
- Reading Order
- Reading and Tab Order in Practice
- Skipping Repetitive Components

Effective application development depends on an understanding of the needs of the user.

- Forms
- Focus
- Color
- Flickering

The examples cited in this document are built using the Flex SDK, and either Adobe Flash Builder 4 or command line tools. When using the command line tools, accessibility options are set by editing a Flex MXML document. In Flash Builder 4, most accessibility properties can be set through the Properties view. Other properties closely related to accessibility, such as `tabIndex`, `focusEnabled`, and `tabEnabled`, can also be set through the same Properties view.

## An overview of assistive technologies

To create an accessible Flex application, developers must understand the needs of users with disabilities, how users with different disabilities interact with an application, and the assistive technologies they commonly use.

Users who are blind typically rely on a keyboard, rather than a mouse, to interact with the computer. These individuals often use screen-reading software, such as JAWS for Windows (JAWS is an acronym for Job Access With Speech) or NVDA Window-Eyes, to access information on the computer. These screen readers and others like them convert electronic text to audio speech. For software applications that are accessible, screen readers can provide summary information about what is on the screen, as well as assist users with navigation within the application. Applications that use graphical elements in addition to text can be made accessible to screen reader users by providing textual equivalents for any graphical element that is used to convey information.

People who have a visual impairment other than total blindness may use screen magnification software, such as ZoomText or MAGic, to enlarge the content on the screen or adjust the contrast between foreground and background colors. These users often adjust the appearance of the mouse pointer, caret, and focus rectangle to make them easier to see.

Users who are deaf or hard of hearing rely on visual cues, including captioning or transcripts, to access auditory feedback and the audio portions of multimedia content. Users with a hearing disability other than complete deafness may use assistive hearing devices to amplify sound, or they may rely on the ability to increase the volume of audio produced by an application.

Users who have speech impairments may not be able to produce speech or may have difficulty producing speech that can be interpreted by voice controlled/activated software applications. Although speech is not required for the vast majority of applications, voice-controlled or voice-activated applications are becoming more common. While voice control features can improve accessibility for users with mobility impairments and other disabilities, it is important to keep in mind that users with speech impairments will need an alternative way to control these applications and enter information into them.

Individuals who have mobility impairments may have difficulty manipulating a mouse or using a standard keyboard. Often, these individuals will use alternative input devices to interact with an application. They may use speech recognition software, such as Dragon Naturally Speaking Professional, to substitute voice commands for keyboard and mouse input. These individuals may also use on-screen keyboards that are activated by pointing devices, hardware switches, or eye gaze technology. Additional assistive technologies such as word prediction software and foot-controlled mice are also common.

## Accessibility testing overview

Assistive technologies obtain information about a Flex application from the Adobe Flash Player instance in which the application is executing. This information is provided via the Microsoft Active Accessibility (MSAA) Application Programming Interface (API). While some accessibility information is recognized and provided automatically via MSAA, Flex developers must take explicit steps to make most accessibility information available to assistive technology. Many of the techniques described in this document are directed at exposing this information to assistive technology.

Ultimately, accessibility is a measure of how well an application can be used by people with disabilities. The most effective tests for accessibility are conducted through testing by individuals with disabilities. Developers

may also wish to test applications directly with assistive technology. For example, developers can install a screen reader on their machine and attempt to complete use cases relying solely on the keyboard for input and with the monitor turned off.

When testing accessibility, it is important to remember that assistive technology is used to interact with all applications running on an operating system, not just the application under test. Assistive technology can monitor events generated by the system, intercept and monitor output from the computer's video card and sound card, and intercept keystrokes and act on them before they reach an application. Assistive technology's purpose is to assist the user, not to be a testing tool for developers. In determining labels for elements, for example, some assistive technology may be able to make *informed guesses* regarding label associations and thereby work adequately with an application even if has not been created according to accessibility standards. In practice, this means that testing with assistive technology may not be sufficient. Often it is difficult for a developer or quality assurance engineer to effectively evaluate accessibility solely using assistive technology.

To address the difficulties inherent in accessibility testing, Adobe recommends the use of AccProbe (*http://accessibility.linuxfoundation.org/a11yweb/util/accprobe/*) or aDesigner (*http://www.eclipse.org/actf/downloads/tools/aDesigner*) to test compiled Flex applications. The tools provide a means of inspecting the MSAA information exported by an application. Developers can use AccProbe or aDesigner to inspect the Flex application as it executes and validate that it is exporting proper information via the MSAA API. This testing approach enables developers to verify that core MSAA information is being made available by the Flex application without direct testing in assistive technology.

**Note**: To display Flex content in an accessible fashion, developers must have the current version of Adobe Flash Player installed. Currently, only the Internet Explorer and Firefox Flash Player plug-ins support the export of accessibility information via MSAA. As such, developers should only perform accessibility testing in one of these two browsers.

## Enabling accessibility

While Flex applications provide a high degree of accessibility automatically, developers must still take a number of explicit steps to ensure that accessibility is enabled at both the application level and the individual component level. These steps ensure that assistive technology can access accessibility information for each application and its components.

In Flex 4, accessibility is enabled for applications by default. In addition, developers can ensure that accessibility is active by updating the Flex compiler setup globally via the command line or for any specific Flex application.

### Enable accessibility in the Flex Compiler

To ensure accessibility is enabled for all Flex applications, edit the flex-config.xml file in the frameworks folder of the Flex SDK directory and define the value of the `accessible` tag of the `compiler` tag to be true as shown in the following example:

```
<flex-config>
  ...
  <compiler>
     ...
     <accessible>true</accessible>
   ...
  </compiler>
</flex-config>
```

### Enable accessiblity in Flash Builder 4

To enable accessibility for a Flex application in Adobe Flash Builder 4, choose Project > Properties, select Flex Compiler, and select the Generate Accessible SWF File option (see Figure 1).

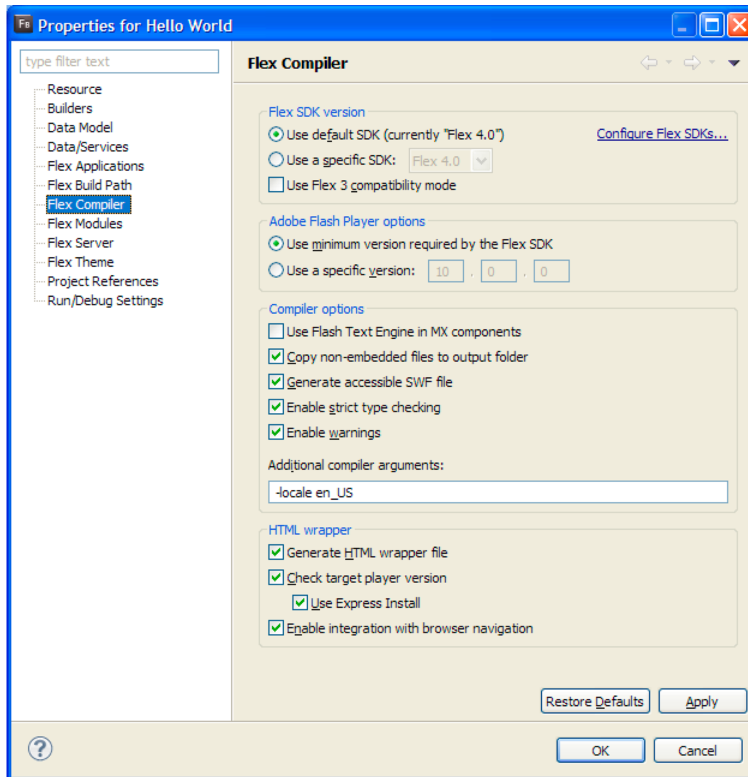


**Figure 1**

The Project Properties dialog box in Flash Builder 4 with the Generate Accessible SWF File setting selected.

### Enable accessibility using the command-line compiler

When compiling a file using the mxmlc command-line compiler, it is possible to explicitly override the setting in the framework-config.xml file. To enable accessibility via the command-line compiler, use the `compiler.accessible` command line flag (or just `-accessible`), as shown in the following example:

```
c:>c:\flex _ 4 _ sdk\bin\mxmlc.exe -compiler.accessible myApp.mxml
```

## Embedding accessible Flex applications

When a Flex application is to be deployed on the web, developers need to properly embed the Flex content into an HTML page to ensure that it interacts in an accessible fashion with the host web page.

**Note**: The default HTML file created by Flash Builder 4 embeds Flex applications properly in the generated HTML. Developers who use the standard embedding code do not need to modify application elements to ensure that they are accessible.

Developers who do not use the standard embedding code must set the Window Mode (`wmode`) parameter of the `OBJECT` element that is used to embed the SWF content in the HTML page to `window`. (This can also be accomplished by ensuring the `wmode` parameter is not set; in this case Flash Player will use `window` as the default value.) To set this value using a `PARAM` element as a child of the embedding `OBJECT` element, use the following syntax:

```
<PARAM NAME="wmode" VALUE="window">
```

Transparent applications that allow the web page to show through SWF content, and opaque applications that obscure but allow the SWF content to be on top of the page, are not accessible by assistive technologies because there is no defined child window created by Flash Player inside of the browser. If the application being developed does not allow for the window mode to be set to `window`—because items must be displayed in front of or behind the SWF content—developers may provide an alternative page that inserts that content using the `wmode='window'` parameter.

Developers must also ensure that users who rely exclusively on the keyboard for navigation do not get trapped in the Flex content. Provide a way to move the keyboard focus out of or past the Flex content. Generally this is achieved by ensuring that the `SeamlessTabbing` parameter of the `OBJECT` element embedding the Flex content is set to `true` (or not set at all, as it will default to true). To set this value using a `PARAM` element as a child of the embedding `OBJECT` element, use the following syntax:

```
<PARAM NAME="SeamlessTabbing" VALUE="true">
```

If an application provides its own focus management, the developer must ensure that the custom focus manager properly handles keyboard focus in and out of the Flex content.

**Note**: In some versions of Firefox, there is a known issue with the Flash Player plug-in that makes it difficult to get focus within SWF content. This may prevent keyboard users from being able to tab into the Flash Player plug-in. If the target audience will be using Firefox, developers should consider creating a shortcut keystroke that would force focus into the Flex content when activated. This can be achieved using JavaScript and the `.focus()` method or by following technique FLASH17 of the W3C WCAG 2.0 techniques for Flash (*http://www.w3.org/TR/WCAG-TECHS/FLASH17.html*).

## Accessible Flex components

Flex 4 includes a variety of components and containers in the Spark and MX component sets that support accessibility. Each component has been thoroughly reviewed and tested for accessibility and interoperability with assistive technology. In developing the default set of components, one goal was to accelerate development of accessible applications. For the complete list of accessible Spark and MX components provided in Flex 4, see "Accessible Components and Containers" in the Adobe Flex 4 documentation (*http://goo.gl/xiTNu*). (A list of the relevant Spark components and containers is also provided later in this document.)

### Accessibility for custom components

It is possible to create custom UI components that meet the different technical and functional accessibility requirements for access by users with disabilities. Detailed instructions for creating such components are beyond the scope of this document. In situations that require a custom accessible component, developers should plan full support for MSAA from the beginning of the development process to ensure the behavior of custom components is consistent with default Flex components and complies with operating system requirements.

Among other requirements, MSAA requires that accessible components must export name, state, and role information. Developers should ensure that this information is exported by their custom objects in conformance to MSAA as implemented in Flash Player. After MSAA support has been added to a control, it is important to validate that the component works with assistive technology. Assistive technologies, most notably screen readers, rarely implement the entire MSAA specification, particularly when operating with a plug-in or ActiveX control such as Flash Player. Some degree of coordination with screen reader vendors or screen reader-specific scripts may be required if a custom-built component deviates from the approach used in the default Flex component set or a new control type is introduced.

Creating an accessible custom component requires detailed knowledge of MSAA and the various properties that must be exposed for the component itself and its child elements. This approach is limited by what MSAA information can flow through Flash Player and is defined in the MSAA specification itself. Not all types of objects have a direct representation in MSAA. For example, MSAA does not provide a specification for how a calendar component should expose accessibility identity information, nor does it allow for explicit association between table header and data cells. In addition, assistive technologies such as screen readers expect certain MSAA structures and properties based on how SWF content has historically been rendered. As a result, some of these technologies may not function as expected with custom implementations.

In practice, this means that while developers can choose to develop custom, accessible components, they should be prepared for significant development costs due to the complex nature of MSAA and the need to make components work with a variety of assistive technologies. **For this reason, developers are strongly encouraged to use the default Flex components that include built-in accessibility support whenever possible**.

## Accessibility properties

Each MX and Spark component that provides accessibility support exports accessibility information via standard ActionScript accessibility properties. For more information, see the ActionScript 3.0 API (*http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html*) and specifically the `flash.accessibility` API (*http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/accessibility/package-detail.html*).

The primary mechanism for defining and updating accessibility information for user interface objects is the AccessibilityProperties class, which exports basic accessibility information from Flex interface elements to assistive technologies.

### Name and description

Two key properties set via an AccessibilityProperties object are name and description:

- **name** — The accessible `name` property in Flex allows assistive technology to identify user interface elements. The name is the core identifying text for an element and is often set to be the label text placed near or on the element or alternative text for a visual element such as an image. **Note**: The value assigned to this `AccessibilityProperties.name` property is not the same as that assigned to an element's name property, which provides a programmatic name for the element. The `AccessibilityProperties.name` value should be a human readable, descriptive name. An accessible name should be concise and provide all required information for the user to identify the component. Role, state, and value information, such as the type of component (button, check box, and so on), should not be included in the accessible name. This information is exposed through the accessibility implementation associated with the component in the accessible role, state, or value methods.
- **description** — The accessible `description` property (set using `accessibilityProperties.description`) provides supplementary detail of the component via a long form description of its use and purpose. This property is a good place to put information such as special instructions on how to interact with the component or an extended description that may be useful in understanding the component.

The accessible name and description fields can be accessed directly in ActionScript via the object. `accessibleProperties.name` and `object.accessibleProperties.description` properties. Flex 3.5 and later, however, provides convenience accessor methods for accessible MX and Spark objects directly through these properties:

`mx.core.UIComponent.accessibilityName` (*http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/mx/core/UIComponent.html#accessibilityName*)

`mx.core.UIComponent.accessibilityDescription` (*http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/mx/core/UIComponent.html#accessibilityDescription*)

In this document, these accessor methods are used whenever possible for simplicity. If an `accessibilityProperties` object does not exist for a component that supports it, then one will automatically be created when using these convenience methods. In addition, using these convenience methods automatically invokes the `Accessibility.updateProperties()` method, which then informs assistive technology to refresh any cached MSAA information. Thus, developers do not need to call `Accessibility.updateProperties()` when using these accessors.

In the following example, the search entry box defines an accessible name and description directly for the TextInput Spark component:

```
<s:TextInput x="36" y="28" width="252" height="21"
     accessibilityName="Search Term"
     accessibilityDescription=”Results displayed below after submit."/>
<s:Button x="296" y="28"
     label="Search"/>
```

The TextInput element above defines both the `accessibilityName` and `accessibilityDescription` properties for the element. The Spark Button, however, simply defines a `label` attribute, and does not define

an `accessibilityName` or `accessibilityDescription`. For the Button element—as with many Flex elements—the accessibility information is automatically set when the proper visual label is provided. In practice, developers generally do not need to provide more information regarding the accessibility of the button, which simplifies any localization or modification of the label in the future.

### Additional properties

In addition to name and description, there are a number of other properties supported by the AccessibilityProperties class:

- **shortcut** — A string indicating the shortcut keystroke associated with a component. Setting this property does not bind the actual keyboard shortcut, but simply exposes the string to assistive technologies. The convenience accessor for this property is `UIComponent.accessibilityShortcut`; for example:

```
// ActionScript
btnSearch.accessibilityShortcut = "Alt+S";
// MXML
<s:Button label="Search" id="btnSearch" accessibilityShortcut="Alt+S" />
```

- **silent** — A Boolean property that controls whether the accessibility implementation for this component is enabled or disabled. If set to `true`, no accessibility information is exposed for this component instance. This can be useful when the component is hidden off-screen or behind another component. The convenience accessor for this property is `UIComponent.accessibilityEnabled`. Note that the Boolean logic for the convenience accessor is the opposite for the actual property. Setting `accessibilityEnabled` to `false` will hide the accessibility implementation for the component instance, whereas setting `silent` to `true` achieves the same effect. By default, accessibility information is not exposed for components that are rendered out of the Flex application's coordinates or for those that have the `visible` property set to `false`.

```
//ActionScript
btnNextPageSearchResults.accessibilityEnabled = false;
// MXML
<s:Button id="btnNextPageSearchResults" label="Next" accessibilityEnabled="false" />
```

- **forceSimple** — A Boolean property that controls whether child objects have accessibility exposed. There is no convenience accessor for this property, as it is typically used in creating accessibility implementations in Flash and is less likely to be used in Flex applications unless a custom accessibility implementation is created.

**Note**: When the `AccessibilityProperties` object's properties are changed, a call to `Accessibility.updateProperties()` is required to inform assistive technologies that a change has occurred and that any cached information should be marked as invalid. If a convenience accessor is used, this method need not be called. Because it can be processor intensive for assistive technology to rebuild an accessible view of the Flex application, developers are encouraged to call this method only after all accessibility property changes for a given action have been made.

If `updateProperties()` is called on a platform that does not support accessibility, unpredictable results may occur. Developers must first check to see if the platform supports accessibility by importing `flash.system.Capabilities` and checking the Boolean `hasAccessibility` property; for example:

```
import flash.system.Capabilities;
...
if (Capabilities.hasAccessibility)
  {
    // set accessibility properties and call updateProperties();
  }
```

## Setting component name information

The primary accessibility requirement for Spark and MX components is properly setting and exporting an accessible name. While this information can be set directly using the `accessibilityName` property (a technique explained above), generally it will be provided automatically by the platform when the developer sets a text label (either on or near the element) or sets a tooltip for the element.

### Text labels provided with elements

For elements that provide a visual label, the visual label will automatically be provided as the accessible name of the element. Examples of such elements include Button, CheckBox, RadioButton, LinkButton, and Menu components. In the following Button declaration, for example, the `accessibilityName` will be "Search" because that is the button's label:

```
<s:Button x="296" y="28" label="Search"/>
```

If the label does not sufficiently describe the purpose of the button, the developer should explicitly set the `accessibilityName` property.

### Text labels positioned near an element

If a text label is positioned near an element, developers can define the text label as part of a parent FormItem container in Flex. The FormItem will export the label of the Flex control it is grouped with. In the following example, the TextInput element is grouped with a "Search" text label.

```
<mx:FormItem label="Search">
<s:TextInput width="252" height="21" accessibilityDescription=
     "Enter a search term into this field."/>
</mx:FormItem>
```

### No text labels present

For items that do not provide a text label, such as images, custom controls, and complex controls, the name of the element can be exported either via the toolTip attribute or directly through the accessibilityName property of the element. When the toolTip for an element is set, the toolTip text will be appended to the accessible name of the element.

```
//Option one - use the accessibilityName attribute
<mx:Image width="60" height="56" source="assets/icecreampint.jpg"
     accessibilityName="Ice Cream Pint"/>
//Option two - use the toolTip attribute
<mx:Image width="60" height="56" source="assets/icecreampint.jpg"
     toolTip="Ice Cream Pint"/>
```

**Note**: When no text label is present for a standard component (or when a label is provided as part of the element itself) do not include "image", "button", or any other description of the role of the component in the accessible name. This information will be announced automatically by screen reader software.

### Setting the accessible name at runtime

If the accessible name changes at runtime or it cannot be set at compile time because the name of an element is unknown, the accessible name can be set using ActionScript and the convenience accessor `accessibilityName`. When the accessor is used, the developer neither needs to verify that an `AccessibilityProperties` object has been created nor call `updateProperties()`, as these are handled automatically; for example:

```
btnSearch.accessibilityName = "Search";
```

If the accessor is not used, those extra steps are required:

```
if (!btnSearch.accessibilityProperties)
  btnSearch.accessibilityProperties = new AccessibilityProperties();
btnSearch.accessibilityProperties.name = "Search";
Accessibility.updateProperties();
```

### Accessible Spark components and containers

The following Spark components and containers (*http://goo.gl/JeZcz*) are accessible:

- ButtonBar component
- Button  component

- CheckBox component
- ComboBox component
- DropDownList component
- List component
- NumericStepper component
- Panel container
- RadioButton component
- RadioButtonGroup component
- RichEdiableText component
- Slider components
- Spinner component
- TabBar component
- TextArea component
- TextInput component
- TitleWindow container
- ToggleButton component
- VideoPlayer component

### Accessible MX Components

The following MX components and containers (*http://goo.gl/Anqtg*) are accessible:

- Accordion container
- AdvancedDataGrid component
- Alert component
- Button component
- CheckBox component
- ColorPicker component
- ComboBox component
- DataGrid component
- DateChooser component
- DateField component
- Form container
- Image component
- Label component
- LinkButton component
- List component
- Menu component
- MenuBar component
- Panel container
- RadioButton component
- RadioButtonGroup component
- Slider component
- TabNavigator container
- Text component
- TextArea component
- TextInput component
- TitleWindow container
- ToolTipManager
- Tree component

## Checking for the presence of assistive technology

In some situations, it can be helpful to first check for the presence of assistive technology using the `flash.accessibility.Accessibility.active` property. For example, when audio or multimedia plays

automatically by default, the application may disable this feature if assistive technology is detected; for example:

```
import flash.accessibility.Accessibilty;
...
if (Accessibility.active)
  {
    // perform action desired when assistive technology is running,
       such as preventing the automatic playing of audio or multimedia
  myVideoPlayer.stop();
  }
```

The `flash.accessibility.Accessibility.active` property detects the presence of software querying for MSAA objects. While this generally indicates the presence of a screen reader, it is not a definitive test because other assistive technologies and software may also query for this information and give the appearance that a screen reader is running. It is also a good idea to wait a few seconds after the application loads before checking this property to ensure assistive technology has had time to query Flash Player.

## Textual equivalents

When an application conveys information visually without the use of text, developers should provide a textual equivalent to assist users of screen reader software.

### Image alternatives

Because screen readers cannot discern the meaning of image or animated graphic elements on the screen, developers must provide a brief text description of these elements. Text equivalents can be provided for groups of components or for individual components within an application.

Flex applications commonly present images using an Image or Loader component. To convey the contents of these components to a screen reader, use the `toolTip` attribute to specify a text equivalent. Tooltip content is also made visible to sighted users as they mouse over the image. Because `toolTip` attributes are not keyboard accessible, developers must ensure that anything conveyed in the image meets the accessibility requirements for color and contrast to ensure that all users can access the information.

```
<mx:Image width="60" height="56" source="assets/icecreampint.jpg"
     toolTip="Ice Cream Pint"/>
```

A longer, more detailed description of an image that would not be appropriate to non-screen-reader users can be provided using the `accessibilityDescription` property for the object; for example:

```
<mx:Image width="60" height="56"
  source="assets/icecreampint.jpg"
  toolTip="Ice Cream Pint
  accessibilityDescription="Our fine Ice Cream Pint provides the perfect
     serving dish for your homemade ice cream treats"/>
```

This technique can be helpful when extra detail is provided within the image itself or in a separate panel. When the description is in a separate panel, it can be difficult for a screen reader user to locate the description elsewhere on the screen, especially when it changes. Use descriptions only when needed, placing unneeded descriptions on images will make an application verbose and tedious to use.

To help screen reader users understand complex graphs and charts, an equivalent DataGrid representation or a link to an HTML page containing a data table can be invaluable. When providing equivalents for charts and graphics, be sure to provide a summary of the chart, including trends and axes, as well as detailed information if the user is likely to desire more than the summary information. Detailed information may include facts about

each bar, pie slice, or line on a graph. When an alternative data table or DataGrid component cannot be used to present this information, it can be provided in text format instead; for example:

```
// ActionScript
imgChart1.accessibilityName = "Q1, East, Units sold 2,000; Q1, South, Units Sold: 1,800;
     Q1 North, Units sold: 1,700; Q1, West, Units Sold: 2,200";
// MXML
<mx:Image id="imgChart1" accessibilityName="Q1, East, Units sold 2,000;
     Q1, South, Units Sold: 1,800; Q1 North, Units sold: 1,700;
     Q1, West, Units Sold: 2,200" />
```

### Notes on image equivalents and use

For buttons with images, describe the action the button triggers rather than the image itself. For example, for a button with a printer icon, "print form" is a better text equivalent than "printer".

Avoid using the terms "image" and "photograph" to describe the element unless it is essential. For any item rendered, assistive technology will indicate the element type to the user, so including these terms in the equivalent text is redundant. For example, when a screen reader encounters an image with "image of a red ball" as alternative text, it will announce "Image: Image of a red ball." In contrast, when the text alternative is simply "Red ball," the rendering in assistive technology will be the far more concise "Image: Red Ball."

Consistency is important when images are used to invoke an action. For example, if an application uses a print button with an image of a printer and the equivalent text "Print", then that same image and equivalent text should be used for print buttons throughout the application.

Developers should not use the same image to represent more than one concept or function, nor should they associate more than one image for the same function.

Images that are used solely for the purpose of decoration convey no meaning and thus do not need a text equivalent. As described above, however, set the `accessibilityEnabled` property to `false` or set the `silent` property of the `AccessibilityProperties` object to `true` to hide such images from assistive technologies.

### Text that conveys additional information through font, size, or color

Content that conveys meaning through size, shape, location, or color must be accompanied by a textual equivalent for that information. In practice, this generally arises when text styling is used to convey information. For example, a text cloud showing frequency of word use through font size requires a text equivalent, such as an HTML table or HTML ordered list that provides the same data. The preferred method of providing this information in a Flex application via the accessible name on the text cloud Image:

```
//ActionScript
imgCloud.accessibilityName = "Favorite Shaved Ice Flavors in Descending order:
     1. Grape, 2. Orange, 3. Lime";
// MXML
<mx:Image id="imgCloud" accessibilityName="Favorite Shaved Ice Flavors
     in Descending order: 1. Grape, 2. Orange, 3. Lime" />
```

Likewise, when text color is used to convey information, provide a text equivalent or rewrite the text. See "Color" on page 44 for more information. For error text that uses a color such as red, the text "error" can be added to the label of the error text field; for example:

```
//ActionScript
txtError.text = "Error: An incorrect search term was entered.";
// MXML
<s:Text id="txtError" text="Error: An incorrect search term was entered." />
```

Adding the word "error" to the text itself instead of just to the accessible name is preferred because the accessible name property is not apparent to users with color blindness or other visual impairments who are

not using a screen reader. If the word “Error” cannot be added to the error text then an image indicating an error should be used and the accessible name of the image should be set to “Error”; for example:

```
//ActionScript
imgError.accessibilityName = "Error";
// MXML
<mx:Image id="imgError" accessibilityName="Error" />
```

When describing user interface controls that use color to convey information, use equivalent text that describes what the color signifies, not the color itself; for example:

```
//ActionScript
// A red X button with an accessible name of "Stop Process"
btnRed.accessibilityName = "Stop Process";
// a green checkmark with an accessible name of "Start Process"
btnGreen.accessibilityName = "Start Process";
// MXML
// A red X button with an accessible name of "Stop Process"
<s:Button id="btnRed" accessibilityName="Stop Process"
     skinClass="IconButtonSkin" iconUp="Red.jpg" />
// a green checkmark with an accessible name of "Start Process"
<s:Button id="btnGreen" accessibilityName="Start Process"
     skinClass="IconButtonSkin" iconUp="Green.jpg"/>
```

In the example above, note that a red X and green checkmark were used instead of more ambiguous shapes, such as a circle or square. Color alone is not sufficient to convey meaning because users with visual impairments—including color deficiencies—may be unable to distinguish the red color from the green. By using color, appropriate icons, and alternative text, the same visual information is provided in multiple ways.

### Convey hierarchy through textual equivalents

The hierarchy of content is often conveyed visually, for example, by using lines in an organizational chart, multiple indents in an outline, or a pyramid shape. Developers of accessible applications must ensure that any hierarchical relationships that are conveyed visually are explained in text.

Flex allows list item tags (`li`) in HTML text, however the hierarchical nature of text in lists constructed using these items is not conveyed to MSAA or the screen reader software. Thus, hierarchical information must be conveyed via an outline structure—such as a Tree or AdvancedDataGrid component—or as a text equivalent.

### Using a Tree component

In Flex, using a Tree component is the recommended approach for providing hierarchical information. The Tree component in the following example conveys a simple organization chart:

```
<mx: Text text="Contact the Appropriate Technical Support Person in Your Department" />
<mx:Tree accessibilityName="Technical Support contacts by Department" width="300"
               height="200" labelField="@label">
  <mx:dataProvider>
    <fx:XML>
      <node label="Senior Management">
        <node label="Tim Springer" data="d0"/>
      </node>
      <node label="Information Technology">
        <node label="Mary Smith" data="d1"/>
      </node>
      <node label="Finance">
        <node label="Ted Robins" data="d2"/>
       </node>
       <node label="HR">
         <node label="Recruitment">
```

```
            <node label="Christopher Sharp" data="d3"/>
          </node>
          <node label="Compensation and Other">
            <node label="David Avila" data="d4"/>
          </node>
        </node>
      </fx:XML>
    </mx:dataProvider>
  </mx:Tree>
```

### Using text equivalents

If a native object cannot be used to convey hierarchical information, a text equivalent for the hierarchy should be provided via the accessibilityName property. For basic hierarchies, a direct description of the hierarchy is sufficient; for example:

```
// ActionScript
imgFoodPyramid.accessibilityName = "The original food pyramid consisted of
four levels each representing a food group.  The large base group Breads and
Cereals appears at the bottom, followed by Fruits and Vegetables, Dairy and
Meat, and the smallest group at the top, Fats and Sweets";
```

```
// MXML
<mx:Image id="imgFoodPyramid" accessibilityName="The original food pyramid
consisted of four levels each representing a food group.  The large base group
Breads and Cereals appears at the bottom, followed by Fruits and Vegetables,
Dairy and Meat, and the smallest group at the top, Fats and Sweets" />
```

For more complex hierarchies, the text equivalent should describe all the elements at each level of the tree. When there are more than two levels in a structure, as in an organizational chart, use the word "level" to indicate each level in the hierarchy. For example, consider the following organizational hierarchy:

Contact the Appropriate Technical Support Person in Your Department

- **Senior Management**
  - Tim Springer
- **Information Technology**
  - Mary Smith
- **Financial**
  - Ted Robins
- **HR**
  - **Recruitment**
    - Christopher Sharp
  - **Compensation and Other**
    - David Avila

The equivalent text for this organization may be defined as follows:

```
imgTechSupportOrgChart.accessibilityName =
   "Level 1 - Senior Management, Level 2 - Tim Springer,
    Level 1 - Information Technology, Level 2 - Mary Smith,
    Level 1 - Finance, Level 2 - Ted Robins,
    Level 1 - HR, Level 2 - Recruitment, Level 3 - Christopher Sharp,
    Level 2 - Compensation and Other, Level 3 - David Avila";
```

### Providing text equivalents for progress bars

Accessible applications must provide a text equivalent for any progress bars that convey important information. An equivalent may not be required for an initial preloader progress bar if loading completes within five seconds, though it is required for all other uses. The Flex 4 ProgressBar control does not provide accessibility support automatically. To provide alternative text for these controls, developers should indicate

the percentage complete (for example, "Content loading: 30% complete") via supplementary text on the screen or the `accessibilityName` property:

```
loadProgressBar.accessibilityName =
"Content loading: " + String(Math.floor(loadProgressBar.value)) + " % complete";
```

### Controlling dynamic and automatically updating content

Developers must provide user controls for any content that automatically updates for a period longer than five seconds. Further, the user controls must be keyboard accessible; see "Keyboard accessibility" on page 15 for more information.

### Decorative and non-essential dynamic content

Decorative and non-essential dynamic content must be accompanied by a control to stop or hide it; for example:

```
btnStop.accessibilityName = "Stop automatic decorative content changes";
btnSop.addEventListener(MouseEvent.CLICK, stopContentUpdates);
function stopContentUpdates(): void
{
  // stop content updates not shown
  // update skin of stop button
  btnStop.accessibilityName = "Display automatic decorative content changes";
}
```

### Automatically advancing sequential content

For news feeds, instructions, and other content that automatically sequences from one item to another, developers should provide controls to pause and step through the content. These controls typically include buttons for Pause/Play, Next, and Previous operations. When the Pause button is activated, it should change to a Play button; for example:

```
// set the accessible names of the story number, icon buttons for pause, next, and previous
// News story panel not shown
txtStoryNumber.text = "Story 2 of 5";
btnPlayPause.accessibilityName = "Pause";
btnNext.accessibilityName = "Next Story";
btnPrevious.accessibilityName = "Previous Story";
btnPlayPause.addEventListener(MouseEvent.CLICK, playPause);
function playPause(e:MouseEvent): void
{
  // set accessible name based on the hypothetical Boolean playing
  if (playing)  // stop playing and change the pause button to play
    btnPlayPause.accessibilityName = "Play";
  else  // play and change the play button to pause
    btnPlayPause.accessibilityName = "Pause";

  // change skin of button not shown
  // advanced news story not shown
}
```

When the Next and Previous buttons are activated, focus can be set to the most appropriate place, such as the news story text itself. If the suspended content is sequential and includes information that should not be skipped, then it should resume at the point at which it was paused by the user. For example, if a user pauses a series of printer connection instructions at step three, then the content should resume at that same step.

### Dynamic real-time content

Real-time content that updates automatically, such as stock tickers, scoreboards, and clocks, creates a unique set of challenges for users with disabilities. Frequent updates can disrupt screen readers and distract users with cognitive disabilities. In Flex applications, some automatically updating content is ignored completely by

assistive technology. As with automatically updated non-real-time content, developers should provide controls to stop or pause automatic updates. With real-time content, however, developers should resume the updates at the current value or time, regardless of when the user stopped it. For example, when a user pauses automatic updates for an auction item's price and the time left in the auction, the current bid price and time remaining should be shown when the user is activates the Play button; for example:

```
btnPlayPause.accessibilityName = "Pause display of automatically updating content
-- this does not stop the auction countdown timer";
btnPlayPause.addEventListener(MouseEvent.CLICK, playPause);
function playPause(e:MouseEvent): void
{
 // set accessible name based on the hypothetical Boolean playing
 if (playing)  // stop playing and change the pause button to play
   btnPlayPause.accessibilityName = "Unpause display of automatically updating content
-- display the current information regarding this auction item";
 else  // play and change the play button to pause
   btnPlayPause.accessibilityName = "Pause display of automatically updating content
-- this does not stop the auction countdown timer";
 // change skin of button not shown
 // If content should update, pull content
}
```

## Keyboard accessibility

An application is considered *keyboard accessible* if users can control it solely through the keyboard. This is a core capability for accessible applications because it is essential for individuals who are blind or visually impaired, as well as those who have mobility impairments and thus have difficulty using a mouse. For an application to be considered keyboard accessible, *all* functionality it offers must be accessible from the keyboard. Any component that can be manipulated via the mouse must also be accessible via the keyboard. The does not imply that keyboard access should replace mouse access; developers should continue to offer multiple methods for using and activating components. Many users with disabilities are unable to operate a keyboard and rely solely on the mouse or pointing devices.

The accessible Flex Spark and MX components support keyboard accessibility by automatically making mouse-defined events accessible via the keyboard. For example, the user can check a CheckBox control, select a List item, and open a ComboBox control using only the keyboard with the default component implementation. As a result, most Flex applications that use accessible components are keyboard accessible, or can be made so with little effort. Keyboard accessibility issues typically arise when developers use custom or inaccessible components, add enhanced mouse functionality to provide one-click access, or move the keyboard focus based on input during form validation.

Components that are not presented in the correct tab order also cause accessibility problems for keyboard-only users. Keyboard accessibility depends on the correct implementation of tab order and shortcut access.

- **Tab Order** — The tab order of an application is controlled by the `tabIndex` component property, which allows developers to control the tab order of all tab-enabled components within an application. This order is the sequence that the keyboard focus follows when a user presses the Tab key; the reverse order is followed when the user presses Shift+Tab. The `tabIndex` property can be defined in MXML or ActionScript for any display objects with which the user can interact (any InteractiveObject control). All `tabIndex` values must be greater than 0 and should not be duplicated in the tab order. The `tabIndex` values do not need to be sequential.
- **Keyboard Shortcuts** — Keyboard shortcuts enable quick access to components or actions via the keyboard. Proper use of keyboard shortcuts is vital to keyboard accessibility because it minimizes the number of keystrokes necessary to access objects. Keyboard shortcuts can be defined in MXML or ActionScript using the `accessibilityShortcut` property for each UIComponent element.

**Note**: While *tab order* and *reading order* are not interchangeable terms, in Flex they are both controlled through the `tabIndex` attribute of an element. Reading order is independent of a component's `tabEnabled` state. See "Reading order" on page 23 for more details.

If a tab order is not specified, a default tab order based on the horizontal and vertical coordinates of each component will be used. Depending on the application and the layout of components on the screen, this default tab order may not be logical. Explicitly setting the `tabIndex` attribute of each component will ensure a logical tab order.

### Tab, Space, Enter, and arrow keys

In Flex applications, the Tab, Shift+Tab, Space, Enter, and arrow keys are the primary keys used to navigate to components and activate them. Developers must ensure their default behavior is correctly supported. The following list describes these primary keystrokes:

- Tab navigates to the next actionable component.
- Shift+Tab navigates to the previous actionable component.
- Space activates a button, checks a check box, manipulates a control, or types a space in a text field or text area.
- Enter activates a button (when the `defaultButton` property has been set) or starts a new line in a text area.
- Arrow keys (up and down) move among the choices in a menu, list component, combo box, drop-down list, or group of radio buttons (selecting them without activating them), and move among the lines in a text area.

There is no need to create event listeners to implement these key presses in Flex. The standard Flex event handler for mouse clicks is also triggered by pressing Space (and Enter if the `defaultButton` property is set). Standard Flex components also handle Tab, Shift+Tab, and arrow keys appropriately. Developers should ensure that the `tabEnabled` property is set on all actionable items that are to be included in the tab order. For standard components, this is handled automatically. For custom components, an event handler is required to process mouse clicks and relevant key presses; for example:

```
// the Mouse Click event will also monitor for the space bar to be pressed
and will be triggered by enter as well if a defaultButton has been set in the container
btnSearch.addEventListener(MouseEvent.CLICK, performSearch);
function performSearch(e:MouseEvent): void
{
  // search not shown
}
```

To allow Enter (in addition to Space and the left mouse click) to activate a button, assign the defaultButton property of a container to one button in the container. Pressing Enter while focus is on any component in the container will activate the default button. This feature works for containers such as the MX and Spark Panel as well as the MX HBox, VBox, and Form containers. It is often used for login forms to enable a user to simply press Enter after typing the username and password.

```
<s:Panel title="Default Button Example">
        <mx:Form defaultButton="{mySubmitBtn}">
            <mx:FormItem label="Username:">
                <s:TextInput id="username" width="100"/>
            </mx:FormItem>
            <mx:FormItem label="Password:">
                <s:TextInput id="password" width="100"
                    displayAsPassword="true"/>
            </mx:FormItem>
```

```
            <mx:FormItem>
                <s:Button id="mySubmitBtn" label="Login"
                    click="submitLogin();"/>
            </mx:FormItem>
        </mx:Form>
    </s:Panel>
```

Adding event listeners to monitor for Enter and Space key press events will not work correctly for all screen readers and should be avoided when possible. Screen readers such as JAWS for Windows, Window-Eyes, and NVDA trap keys such as Enter and Space and will not send them through to the Flex application in commonly used screen-reading modes.

### When not to provide keyboard access

In general, all operable controls in a Flex application must be keyboard accessible unless one or more of these is true:

- The control is disabled
- The functionality is duplicated by a second, keyboard-accessible component
- The functionality cannot be implemented without a mouse, drawing pad, or other pointing device. (This may include, for example, freehand or pressure sensitive drawing. It does not, however, include drag-and-drop operations, which can often be accomplished through keyboard access and are discussed below.)

Keyboard access can be removed from an item that is typically in the tab order by setting the `tabEnabled` property to `false`. This is done automatically when the `enabled` or `visible` property of the component is set to `false`. The `tabEnabled` property should be used when the component should not receive keyboard focus but must remain enabled and visible.

```
// ActionScript
btnDelete.tabEnabled = false;

// MXML
<s:Button id="btnDelete" tabEnabled="false" />
```

While `tabEnabled` is the preferred way to disable keyboard focus, developers may also use `focusEnabled` for the same effect.

```
//ActionScript
btnDelete.focusEnabled = false;

// MXML
<s:Button id="btnDelete" focusEnabled="false" />
```

When components are grouped together and the whole set of components should not be tab-enabled, the `tabChildren` property can be set to `false`. This property is useful when a title window or simulated modal dialog appears and components in the background need to be removed from the tab order.

```
//ActionScript
vgSignInForm.tabChildren = false;

// MXML
<s:Button id="vgSignInForm" tabChildren="false" />
```

### Keyboard accessibility considerations

When ensuring the keyboard accessibility of an application, the following cases merit special attention:

### Drag-and-drop functionality

Many users who have difficulty using a mouse will be unable to use applications that depend upon drag-and-drop functionality. Hence, developers must provide a keyboard-accessible way to perform drag-and-drop

actions. For example, if a user can drag a product into a shopping cart, pressing the Enter key on an "add to cart" image or link may be one way to perform the same task. Implementing keyboard shortcuts for cut-and-paste functionality is another technique used to make a common drag-and-drop operation accessible. When possible, use the simplest method that requires the fewest keystrokes. This implies that links are preferable to Image components. Since a link is keyboard accessible and is capable of displaying an image as an icon, it accomplishes the same goals as the Image component, which may not have keyboard access defined. In addition, a redundant event can easily be assigned to the link that moves the product to the shopping cart; for example:

```
lnkAddToCart.addEventListener(MouseEvent.CLICK, addToCart);
function addToCart(e:MouseEvent): void
{
  // add to cart functionality goes here
}
```

### Text

In some cases, uneditable text should be included in the tab order, to provide easy access to it for screen reader users. Alerts and error messages, license text, or any text that is dynamically updated may fall into this category. Placing this text in the tab order allows users of screen readers to tab to the component and have the current value of the text read to them. Unfortunately, the Text and Label components cannot simply be placed in the tab order by setting a tab index and tab enabling them. Flex Text and Label controls are not included in the tab order by the FocusManager class. There are two commonly used techniques for handling this situation. First, developers can simply use a TextInput component and set the `editable` property to `false`; for example:

```
// ActionScript
txtTimeLeft.editable = false; // ensure that the user cannot change
   the value of the TextInput
txtTimeLeft.tabIndex = 13; // set the tab order for this element
txtTimeLeft.tabEnabled = true;  // should be enabled by default but
   this will explicitly set it

// MXML
<s:TextInput id="txtTimeLeft" editable="false" tabIndex="13" tabEnabled ="true" />
```

The second technique is to extend the standard Label or Text component class and ensure the extended class implements the Focus Manager Component interface (IFocusManagerComponent); for example:

```
// create a custom class, FocusableLabel.as
package
{
  import spark.components.Label;
  import mx.managers.IFocusManagerComponent;

  // implement the focus manager
  public class FocusableLabel extends spark.components.Label implements
      IFocusManagerComponent
  {
    // the constructor
    public function FocusableLabel()
    {
      super();
    }
  }
}
```

The example below illustrates how the FocusableLabel class defined above can be used:

```
//ActionScript
import FocusableLabel;
...
var txtTimeLet: FocusableLabel = new FocusableLabel();
txtTimeLeft.text = String(getTimeLeft());  // getTimeLeft function not shown
// positioning of txtTimeLeft not shown
accChild(txtTimeLeft);
txtTimeLeft.tabIndex = 13; // set the tab order for this element
```

**Note**: As described in "Controlling dynamic and automatically updating content" on page 14, developers should provide keyboard-accessible controls that allow the user to pause, hide, or stop automatically updating text.

## Rollover content

Meaningful content that appears as the mouse rolls over an element must also be accessible via the keyboard. Generally, this is achieved by ensuring that the user can tab to the item that activates the rollover, that Enter or Space can activate the rollover content, and that the now visible rollover content itself can be reached via the Tab key even if the content is not actionable.

If the rollover content is not extensive, it can be placed in an accessibility property (for example, `accessibilityDescription`), which will enable users to access it with assistive technology without having to activate the rollover. Support for triggering the rollover via the keyboard is still required, however, as this property is not readily available to users who do not have screen readers. The following example illustrates how an event listener can be used to show rollover content:

```
btnActuator1.addEventListener(MouseEvent.CLICK, showRollOver);
btnActuator1.tabIndex = 10;
btnActuator2.addEventListener(MouseEvent.CLICK, showRollOver);
btnActuator1.tabIndex = 12;
btnActuator3.addEventListener(MouseEvent.CLICK, showRollover);
btnActuator1.tabIndex = 14;


function showRollOver(e:MouseEvent): void
{
  switch (e.target)
  {
    case btnActuator1:
    {
      lblRollOverContent.text = "Content for Rollover 1";
      lblRollOverContent.visible = true;
      lblRollOverContent.tabIndex = 11;
      lblRollOverContent.setFocus();
    }
    case btnActuator2:
    {
      lblRollOverContent.text = "Content for Rollover 2";
      lblRollOverContent.visible = true;
      lblRollOverContent.tabIndex = 13;
      lblRollOverContent.setFocus();
      // display the rollover content for the second actuator
    }
    case btnActuator3:
    {
      // display the rollover content for the third actuator
```

```
      lblRollOverContent.text = "Content for Rollover 3";
      lblRollOverContent.visible = true;
      lblRollOverContent.tabIndex = 15;
      lblRollOverContent.setFocus();
    }
  }
}
```

The example above does not show how the rollover content is hidden, but this can be achieved using the `FOCUS _ OUT` event; for example:

```
lblRollOverContent.addEventListener(FocusEvent.FOCUS _ OUT, hideRollOver);


function hideRollOver(): void
{
   lblRollOver.visible=false;
}
```

Testing should be performed by users with disabilities to determine the best method of hiding rollover content. For example, if the content disappears when focus leaves the content, text would need to be added to the accessible name of the actuator explaining this.

### Context menus

Context menus, which open with a right-click (Windows) or a Control-click (Mac), change depending on the item clicked. In an accessible application, this functionality must also be available from the keyboard. Consider using a shortcut key (such as Shift+F10 on Windows) or an alternative that is accessible from the keyboard (such as a small button next to the item) to activate the context menu.

### Toolbar functionality

If a toolbar cannot be made keyboard accessible, add shortcut keys or menu items to perform the same actions. Anytime keyboard shortcuts are used, clearly document the shortcuts for the user in an accessible format. The documentation should be in the Flex application itself or in the HTML page displaying it, but it may also be provided in a user guide or other support documentation.

See "Shortcuts and mnemonics" on page 22 for more details on creating and exposing keyboard shortcuts.

### Navigation between panes

For applications with two or more panes, keyboard navigation between panes may be needed. This can be provided via a menu, shortcut keys, or the Tab key, if tabbing through all elements in one pane leads to the next pane. On Windows, for example, F6 is a commonly used shortcut key for pane navigation. For application panes with few components, tabbing through all of the components to get to the next pane is acceptable. When panes contain many items however, particularly repetitive components, developers should include a mechanism to quickly move focus past the current pane to the next pane. This allows keyboard-only users to effectively access content in any pane.

See "Implementing keyboard shortcuts" on page 22 for examples on setting keyboard shortcuts.

### Resizing, moving, and closing windows

If a window or component can be resized, moved, or closed with the mouse, a user must be able to perform those same actions using the keyboard. For example, if an object can be moved via the mouse after choosing a menu item, ensure that the menu and menu item are keyboard accessible and that the arrow keys can then be used to move the object. Alternatively, provide input fields to allow the user to manually enter new x, y, width, and height values for the object.

The following example code is from an application that displays a photograph and allows the user to crop the picture. The application provides a button, menu option, or shortcut keystroke to activate the crop feature. It

includes an event listener that moves the crop area when specific keys are pressed. The Control+Shift+arrow keys control the crop rectangle's location.

```
imageCropArea.addEventListener(KeyboardEvent.KEY _ DOWN, adjustCrop);
function adjustCrop(e:KeyboardEvent): void
{
  // check to ensure shift and control are pressed
  if (!e.ctrlKey || !e.shiftKey)
    return;
  switch (e.keyCode)
  {
    case 37: // left arrow
    {
       // move crop area left
    }
    case 38: // up arrow
    {
      // move crop area up
    }
    case 39: // right arrow
    {
       // move crop area right
    }
    case 40: // down arrow
    {
      // move crop area down
    }
}
```

### Keyboard access to scrolling content and fields

In a Flex application, if a component in a scrollable area receives focus when it is not within the scrollable view, it is not automatically scrolled into view when the user tabs to it. It is best, therefore, not to use scrolling areas that contain actionable elements. If active elements must be used within scrolling areas, see "Creating Tab Order and Reading Order" (*http://goo.gl/nwLJv*) in the Flex 4 documentation for an example that shows how to scroll to a component when tabbing. Screen readers may not detect components that were previously out of view fast enough when they are brought into view. See "Sending events" on page 43 for details on how to use the `sendEvent()` method of the Accessibility class to notify the screen reader software that a new component has received focus.

When lengthy non-actionable text (such as license text) is placed in a field, the field must be scrollable via the keyboard so keyboard-only users can review it. By default, some components provide keyboard access via the up arrow, down arrow, page up, and page down keys. Developers should verify this functionality for the component, and, if it is not provided by default, implement another keyboard-accessible scrolling mechanism.

### Elements that should not be keyboard accessible

Some elements should not be keyboard accessible. Including the following elements in the tab order only slows down access to content by users who rely solely on the keyboard for navigation:

- Form labels. Users need to be able to tab to form elements, not to their labels.
- Elements not visible on the screen, including elements that are offstage or hidden behind other elements. Note that this does not include actionable controls that are out of sight in a scrollable area.
- Elements that are inactive or disabled, unless it is important to convey that the button exists. In that case, the state of the button should be conveyed if the focus is set on that disabled object.
- Decorative objects that convey no meaningful information.

Placing an element behind another in the z-order (front to back order) to hide it is not recommended, as the element may still appear in the tab order. Thus, developers should be sure to set the visible property on items that are not `visible` to `false`, move elements that are off-screen completely off-screen, and ensure that `tabEnabled` is set to `false` when an element is visible but should not appear in the tab order.

### Shortcuts and mnemonics

Frequently used functions of an application should be accessible via keyboard shortcuts, so that they can be accessed as quickly with the keyboard as with a mouse. Keyboard shortcuts typically use an easily remembered letter (or *mnemonic*) from the name of the component or action. The letter is underlined in that name to help users identify and remember it. For example, Ctrl+S is a standard mnemonic for a Save button or Save menu item.

There are no specific guidelines that explain when a shortcut is required, but it is a good practice to add a shortcut for any commonly used component that:

- Requires pressing the Tab key more than five times to reach
- Controls media playback
- Is needed quickly, such as to silence audio or stop repeated content updates

It is important to indicate shortcuts on the screen. Though it is possible to indicate them via the `accessibilityShortcut` property, screen readers do not always make use of this property and the information provided there will be unavailable to keyboard-only users who are not using a screen reader. In addition to underlining shortcut letters in menu items and button names, it is often helpful to make a list of shortcuts available on the screen or in a keyboard-accessible pop-up window. A list of shortcut keystrokes can also be provided in user guides and online help documentation.

### Keystrokes to silence audio

It is very difficult for users of a screen reader to listen to an audio track and the screen reader at the same time. Audio that plays automatically when an application is loaded is especially disruptive. Thus, developers must ensure that users of assistive technology can stop the playback of any audio content (including narration, multimedia, or background music) that lasts longer than five seconds.

In the following example, Ctrl+S is used to pause the audio content:

```
addEventListener(KeyEvent.KEY _ DOWN, pauseVideo);
function pauseVideo(e: KeyEvent): void
{
  // if ctrl+s is pressed pause the multimedia presentation
  if (e.ctrlKey && e.keyCode == 83)
    myVideoPlayer.pause();
}
```

### Implementing keyboard shortcuts

Event listeners for keyboard shortcuts should be placed on the most general component in the application or on the application itself. If the event listener is attached to a specific component, the shortcut keys it handles will only be active when the focus is on that component.

As an example, consider a developer who wants to provide quick access to the help screen for an application by using the question mark (?) key as a shortcut. The developer would add a listener for the `KEY _ DOWN` event on the main application object as shown in the following example:

```
// assign the event listener directly on the main application object
// For applications that accept user input (such as into form fields) the
 function below should be designed to verify that the user is not current
 in a form field.  Otherwise a keystroke such as F1 or a keystroke
 combination such as Control+Shift+? could be used rather than the
 single key question mark (?).
```

```
addEventListener(KeyEvent.KEY _ DOWN, performShortcut);

function performShortcut(e:KeyEvent): void
{
  if (e.keyCode == 63)  // the question mark (?)
  {
    // call same code that button's click handler calls
  }
}
```

### Shortcuts to avoid

When choosing shortcut keystrokes for a Flex application, try to avoid the main shortcuts used in browsers, operating systems, and assistive technologies. The following shortcuts may cause difficulty if used in a Flex application:

- Windows: F1 (help), Alt+F4 (close the application), Ctrl+F4 (close the current window), Ctrl+W (close the current window), Ctrl+Tab (switch tabs)
- Internet Explorer and Firefox: F11 (full screen), Alt+D (place the focus in the address bar), Ctrl+P (print), Ctrl+N (new window), Ctrl+T (new tab), Ctrl+F (find), Ctrl+D (Bookmark), Ctrl+H (History)
- JAWS and Window-Eyes: Numpad Plus (toggle forms mode), Ctrl+Shift+A (toggle MSAA mode), Ctrl+Home (move to top of document), Ctrl+End (move to bottom of document), most single letters, Shift+most single letters, Insert+Down Arrow (JAWS, say all), Control+Shift+R (Window-Eyes, read to bottom of document))

While some conflicts are unavoidable, choose shortcuts carefully to avoid those most commonly used.

**Note**: Screen readers trap most single letter keystrokes in virtual cursor or browse mode. When this mode is activated, Flex applications will not receive keystrokes until the user activates forms mode or MSAA mode, or passes the keystroke through to the application by pressing another keystroke combination.

## Reading order

Providing textual equivalents, enabling, keyboard access, and setting accessibility properties is not, by itself, enough to make an application accessible to users of all assistive technology. Developers must also take steps to ensure that content is exposed to assistive technology in a sequence that is meaningful. This sequence is known as reading order and describes the order in which a screen reader reads the content of the screen. The default reading order of any SWF file, whether created in Adobe Flash or with Flex, does not always follow a predictable left-to-right, top-to-bottom order. When the content itself does not explicitly specify a reading order, Flash Player determines the default order based on a formula that uses the x and y coordinates of a component's bound rectangles. For simple Flex applications, reading order may not be an issue; however, for most Flex applications, the reading order should be explicitly specified. If it is not, the content may be difficult or impossible to understand with a screen reader.

The example code below defines a tree component, a data grid, and a text area. In this example, the default reading order flows from the tree control to the data grid and then to the text area (see Figure 2). This is the order in which the items will be encountered by screen reader users, and it is the desired order for the application. Developers can confirm this default order by reviewing the content with a screen reader or testing it in AccProbe. In this case, the screen reader would announce the browser page title, followed by the tree component, the data grid, and the text area. Since this is roughly how a user would follow the application visually, the reading order does not need to be set explicitly.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955"
  minHeight="600">
  <mx:Form>
    <mx:Tree accessibilityName="Lunch Items:" width="300" height="150"
```

```
        labelField="@label">
<mx:dataProvider>
  <fx:XML>
    <node label="Foods">
      <node label="Fruits">
        <node label="Apples" data="apples"/>
        <node label="Oranges" data="oranges"/>
        <node label="Bananas" data="bananas"/>
      </node>
      <node label="Veggies" data="veggies"/>
      <node label="Desserts" data="desserts"/>
    </node>
  </fx:XML>
</mx:dataProvider>
    </mx:Tree>
    <mx:DataGrid accessibilityName="Pets:" height="150" id="fg" minWidth="200">
    <mx:columns>
      <mx:DataGridColumn headerText="ID" dataField="cid"/>
<mx:DataGridColumn headerText="Name" dataField="name"/>
<mx:DataGridColumn headerText="Age" dataField="age"/>
    </mx:columns>
    <mx:ArrayList>
<fx:Array>
  <fx:Object cid="1" name="Arnold" age="3" />
  <fx:Object cid="2" name="Denver" age="5" />
  <fx:Object cid="3" name="Jack" age="8" />
</fx:Array>
    </mx:ArrayList>
    </mx:DataGrid>
    <s:TextArea accessibilityName="Comment" heightInLines="3"
          id="ta1" text="Enter text" />
  </mx:Form>
</s:Application>
```

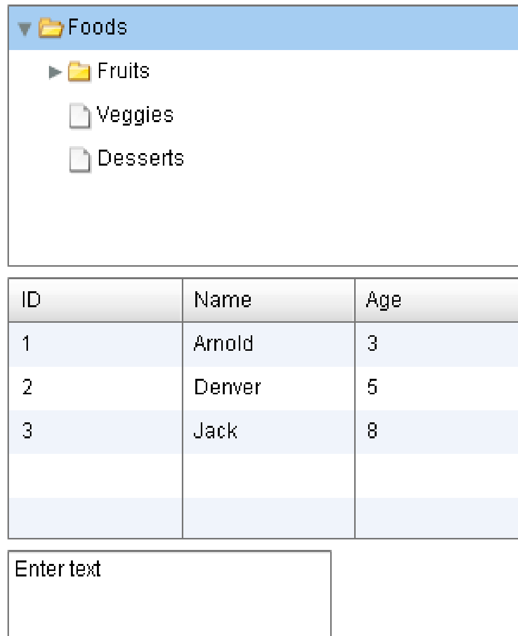


**Figure 2**

**A Flex application with three components: a tree view, a data grid, and a text area.**

The next example presents a slightly more complex application (see Figure 3). In this case, it is possible that labels associated with specific controls will not be announced by screen reader software with their corresponding text input control. Thus, users may believe that they should type the ZIP code in a field when, in actuality, they should type their telephone number. In applications like this, the developer must control the reading order using the `tabIndex` attribute.

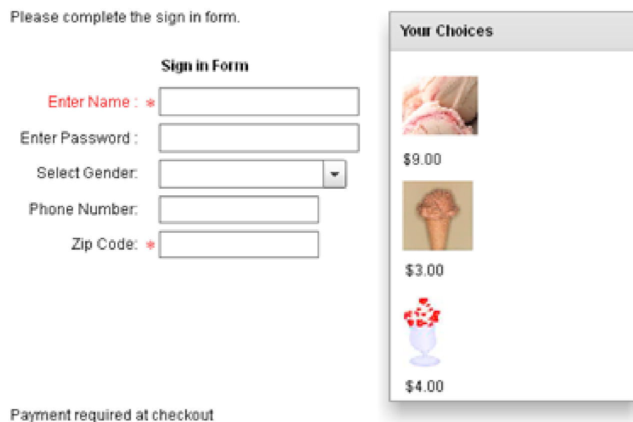


**Figure 3**

The sign in form for a shopping cart application that displays selected products.

More complex layouts, like the one used in the application in Figure 3, are more likely to cause issues for screen reader users. The application uses a repeater to populate a grid. There are three rows of products, and each product has an image and a label indicating the cost of the product. Since the labels and images are in separate components, they may not necessarily be read one after the other in a predictable order when the default Flash Player reading order is used. Instead, this application controls the reading order itself. Here is the code:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx"
```

```
                minWidth="955" minHeight="600">
<fx:Declarations>
      <!-- Place non-visual elements (e.g., services, value objects) here -->
</fx:Declarations>
<mx:VBox>
<mx:HBox>
<mx:VBox>
<s:RichText paddingLeft="10" paddingTop="10" tabIndex="1"
     text="Please complete the sign in form." width="287" />
<mx:Form>
      <mx:FormHeading label="Sign in Form" tabIndex="2"></mx:FormHeading>
      <mx:FormItem id="f1" tabIndex="3" color="#ff0000" required="true"
           label="Enter Name :">
         <mx:TextInput id="txt1" tabIndex="4" />
      </mx:FormItem>
      <mx:FormItem label="Enter Password :" tabIndex="5">
        <mx:TextInput displayAsPassword="true" tabIndex="6"/>
      </mx:FormItem>
      <mx:FormItem label="Select Gender:" tabIndex="7">
        <s:ComboBox id="c1" width="150" tabIndex="8">
                <s:dataProvider>
                        <mx:ArrayList>
                                <fx:String>Male</fx:String>
                                <fx:String>Female</fx:String>
                        </mx:ArrayList>
                </s:dataProvider>
        </s:ComboBox>
      </mx:FormItem>
      <mx:FormItem label="Phone Number:" tabIndex="9">
        <s:TextInput id="txtPhoneNumber" tabIndex="10" />
      </mx:FormItem>
      <mx:FormItem required="true" label="Zip Code:" tabIndex="11">
        <s:TextInput id="txtZipCode" tabIndex="12" />
      </mx:FormItem>
      <s:Button id=”submit” label="Proceed to Checkout" tabIndex="13" />
</mx:Form>
</mx:VBox>
    <mx:VBox paddingTop="10">
      <s:Panel title="Your Choices" width="200" tabIndex="20" >
        <s:layout>
                <s:VerticalLayout paddingTop="20" paddingLeft="10" gap="10"/>
        </s:layout>
                <mx:VBox>
                        <mx:Image width="60" height="56" source="assets/icecreampint.jpg"
                             toolTip="Pint of Ice Cream" tabIndex="21"/>
                        <s:Label text="$9.00" tabIndex="22"/>
                </mx:VBox>
                <mx:VBox>
                        <mx:Image width="60" height="56" source="assets/iceCreamCone.jpg"
                             toolTip="Ice Cream Cone" tabIndex="23"/>
                        <mx:Label text="$3.00" tabIndex="24"/>
                </mx:VBox>
                <mx:VBox>
                        <mx:Image width="60" height="56" source="assets/sundae.jpg"
                             toolTip="Ice Cream Sundae" tabIndex="25"/>
                        <mx:Label text="$4.00" tabIndex="26"/>
```

```
                </mx:VBox>
            </s:Panel>
</mx:VBox>
    </mx:HBox>
<s:Label paddingLeft="10" text="Payment required at checkout" tabIndex="27"/>
</mx:VBox>
</s:Application>
```

The code above explicitly sets the reading order by assigning values to the `tabIndex` attribute for each component via MXML (`tabIndex` can also be set via ActionScript).

### Reading order best practices

There are several best practices to keep in mind when defining the reading order for an application.

### Be logical

For anything but simple Flex applications, explicitly define the reading order so that screen reader users hear the contents of a page in a logical, intuitive way—usually the same way a sighted user would read the page. Since the same property (`tabIndex`) is used to control both reading order and tab order, setting `tabIndex` appropriately to specify a logical reading order will also enable keyboard-only users to tab through a page without excessive jumping around and to follow form fields in a way that makes sense. See “Keyboard accessibility” on page 15 for more information.

### Order error messages first

Error messages that pertain to an entire form should be read before the form itself, so they need to be placed at the top of the form and first in the reading order. Field-specific error messages should be provided inline (appended to the component’s accessible name) and at the beginning of the form; for example:

```
// ActionScript
txtFormError.tabIndex=2;
// other form fields here
txtPhoneNumberError.tabIndex=9;
txtPhoneNumber.tabIndex=10;

// MXML
<s:Text id="txtFormError" tabIndex="2" />
<s:Text id="txtPhoneNumberError" tabIndex="9" />
<s:TextInput id="txtPhoneNumber" tabIndex="10" />
```

See “Forms” on page 32 for more information on providing accessible error messages.

### Disable accessibility for decorative and non-relevant content

All purely decorative content should be hidden from assistive technology. When nonessential content is not presented to users of assistive technology such as screen readers, the important content is easier to access.

Likewise, content that is hidden behind a modal TitleWindow layout container or simulated dialog box should be made inaccessible. Modal windows require the user to interact with them when they appear on top of other content, so the accessibility of all elements in the background should be disabled. When the simulated dialog or modal TitleWindow is closed, accessibility should be enabled for the components once again. For example, when a help button is activated and a TitleWindow appears with help content in it, set `accessibilityEnabled` to `false` for all of the content in the background, including the Help button:

```
// turn off accessibility for the Help button
btnHelp.accessibilityEnabled = false;
// turn off accessibility for all other background content --
     in this case in a VGroup component and the components contained in it
vgMainGrouping.accessibilityEnabled = false;
```

Only the content in the TitleWindow container should have accessibility enabled.

If accessibility cannot easily be disabled by grouping, developers can place all background components in an array and cycle through the array to disable accessibility for each component individually.

**Note**: In addition to disabling accessibility for hidden elements, keyboard access should also be restricted to current, active elements. See "Keyboard accessibility" on page 15 for information on how to remove keyboard access from hidden content.

### Exclude hidden or inactive elements

Elements that are hidden behind other elements, provided only for decoration, or located off-screen should be excluded from the reading order while they are invisible or inactive. Any active elements that can still be seen by sighted users should remain in the reading order, even if they are read-only (not editable) or disabled. Hidden elements include rollover content that has not been activated and elements that are hidden behind modal pop-up windows. In addition, inactive decorative elements that do not convey information should not be included in the reading order.

Setting the `accessibilityEnabled` property to `false` on a form, grouping, or container component will disable accessibility for all child components; for example:

```
// turn accessibility support off for this form when it is hidden behind a modal pop-up
frmSignInForm.accessibilityEnabled = false;
```

When the property is set on an individual component, the setting affects the accessibility of just that particular component. When accessibility is disabled for a component, it will not be exposed to assistive technology via MSAA.

### Provide controls for disruptive elements

Screen readers may interpret any automatically updating content on the screen as a change that must be reported to the user. This can cause the screen reader to start over at the top of the page every time there is a change on the screen. Examples of elements that can be disruptive to screen reader users when accessibility is not implemented correctly include:

- Animations
- Weather and progress updates
- Clocks
- Scoreboards
- Timers

It is not a good practice to remove these components from the reading order to prevent screen readers from announcing the information repeatedly. This approach makes any dynamic content inaccessible to screen reader users, and does nothing for other users who may simply want to stop, pause, or hide the dynamic content. See "Controlling dynamic and automatically updating content" on page 14 for guidelines on handling such content.

### Avoid redundant elements

When screen reader software encounters a Flex component, such as a form field, it announces the component's accessible name. Screen reader users do not require visual labels for form fields that have an accessible name property, but other users do need the visual label. A problem arises for screen reader users when both the visual label and the accessible name are exposed to assistive technology. In these cases, the screen reader reads both. If the label and accessible name are identical, the user hears everything twice; if they are not identical, the results can be even more confusing. In Flex, this is likely to happen with TextInput, ComboBox, List, DataGrid, and TextArea components. To avoid this redundancy, it is best to hide the visual label for such components from the screen reader by setting the `accessibilityEnabled` property to `false`; for example:

```
lblFirstName.accessibilityEnabled = false;
```

### Silence form items and form headings

In Flex 4, FormItem and FormHeading components expose accessibility properties; prior to Flex 4 they were not exposed by default. Because of this change, label content present in the FormItem containers and

FormHeading control are placed in the reading order along with the form fields. This can cause field labels to be read twice by screen readers because both the FormItem label and the component inside the FormItem have the same accessible name and, by default, both are read. To address this, silence the FormItem container's label by setting the `accessibilityEnabled` property to `false` or the `silent` property to `true`; for example:

```
// where formItem1 is the form item that should be made silent to assistive technology
// check to see if accessibility properties is present for the FormItem's label
     and create it if not
if (formItem1 && formItem1.itemLabel && !formItem1.itemLabel.accessibilityProperties)
  formItem1.itemLabel.accessibilityProperties = new AccessibilityProperties();
//Set the label to be silent - indicating it should not be read in a screen reader
if (formItem1 && formItem1.itemLabel)
  formItem1.itemLabel.accessibilityProperties.silent = true;
```

When a form contains many FormItem containers, developers can populate an array with each FormItem container, loop through the array, and set `silent` property to `true` for each item in the array; for example:

```
// array of all form items to be used to set the accImpl to silent
var col:ArrayCollection = new ArrayCollection
     ([f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,f17]);
var i:int = 0;
// loop through all form items and set the form item label to silent
i = 0;
while (col && i < col.length)
{
  if (col[i] && col[i].itemLabel && !col[i].itemLabel.accessibilityProperties)
    col[i].itemLabel.accessibilityProperties = new AccessibilityProperties();
  if (col[i] && col[i].itemLabel)
    col[i].itemLabel.accessibilityProperties.silent = true;
  i++;
}
```

The FormHeading control is automatically added to the accessible name of each item in the form. Thus, as with the FormItem container's label component, the FormHeading should also be made silent. The FormItem component itself could not be made silent, because it is a container for the form fields and disabling accessibility on it would disable accessibility on its child components. In contrast, the FormHeading control has a direct accessibility representation, so developers can quickly set it to silent without having to access any child components; for example:

```
// set the accImpl on the form heading (fh) to silent
if (!fh.accessibilityProperties)
fh.accessibilityProperties = new AccessibilityProperties();
fh.accessibilityProperties.forceSimple = true;
fh.accessibilityProperties.silent = true;
```

### Notes on hiding content

It is important to remember that content hidden by setting `accessibilityEnabled` to `false` is only hidden from certain classes of assistive technologies—most commonly screen readers. This technique does not hide the content from sighted users who may have other disabilities. Thus, content hidden in this way must still comply with other accessibility guidelines.

## Reading and tab order in practice

Both tab order and reading order are controlled by setting the `tabIndex` property. All components that have `tabEnabled` or `focusEnabled` set to `true` will receive keyboard focus in the order defined by the `tabIndex` property. Likewise, all components that have `enableAccessibility` set to `true` (or `silent` set to `false`) will be read by screen reader software in that same order.

In many cases, there will be more elements in the reading order than in the tab order because screen reader users must access not only keyboard-accessible controls and form fields but also text and images that do not receive keyboard focus. Flex application developers should define `tabIndex` values for all objects that are to be read, including text fields and other non-focusable interface controls.

In practice, there are two main strategies for defining `tabIndex` values for components in an application. For applications in which the overall reading and tab order can be defined at compile time, developers can set the order using static `tabIndex` values. In this case, each component that is to be read explicitly defines a `tabIndex` value in its MXML definition. When manually setting values it is often a good idea to leave space for future components. For example, use `10`, `20`, `30`, `40`, and `50` as `tabIndex` values rather than `1`, `2`, `3`, `4`, and `5`. If additional fields are later needed between the first and second components, for example, they can be assigned `tabIndex` values of `14` and `16`, and the existing values will not need to be renumbered. To ensure a consistent, logical order, use `tabIndex` values that are greater than zero and do not use the same value more than once in the order.

For complex applications in which the number and placement of components is not known at compile time, often a better strategy is to place all of the components in order in an array, collection, or vector and update the `tabIndex` property via ActionScript. When new components are added, the application inserts them into the collection at the proper location, and updates the `tabIndex` property for each element in the collection. A similar approach is used for deleting or reordering components. This approach is easy to scale and can be more convenient than reconciling the order of individual components at compile time.

## Addressing reading order issues with compound components

Many compound components, such as Accordion, TabNavigator, and Panel containers, comprise several visual components. Accordion components, for example, contain header components. When the `tabIndex` property is set on the Accordion container, the change does not automatically propagate to the individual headers. The header components are likely to appear in the incorrect reading order if their individual `tabIndex` properties are not set. Developers must explicitly set the `tabIndex` property on child components for the following components:

- MX Accordion
- MX TabNavigator
- MX Panel

The following example illustrates one way to set the reading order for children of compound components. The `init()` function, which is called when the application is initialized, loops through each child of the Accordion container and sets the `tabIndex` property on the child header to the value of the Accordion container's `tabIndex` plus the position of the child. In this example, the Accordion container has a `tabIndex` of `50`, so the first child header is assigned a `tabIndex` of `51`, the second `52`, and so on.

```
...
<fx:Script>
<![CDATA[
function init(): void
{
  // set the tabIndex of accordion headers so they appear in the correct reading order

  var i:int = 0;
  while (i < a1.numChildren)
  {
    if (a1.getHeaderAt(i))
      a1.getHeaderAt(i).tabIndex = a1.tabIndex+i+1;
    i++;
  }
```

```
}
]]>
</fx:Script>
...
<mx:Accordion id="a1" tabIndex="50" width="100%">
  <mx:Form tabIndex="51" label="Accordion Pane 1" width="100%">
    <mx:FormItem id="f15" label="Text Input 1:">
<s:TextInput tabIndex="54"/>
    </mx:FormItem>
  </mx:Form>
  <mx:Form tabIndex="52" label="Accordion Pane 2" width="100%">
    <mx:FormItem id="f16" label="Text Input 2:">
<s:TextInput tabIndex="55"/>
    </mx:FormItem>
  </mx:Form>
  <mx:Form tabIndex="53" label="Accordion Pane 3" width="100%">
    <mx:FormItem id="f17" label="Text Input 3:">
<s:TextInput tabIndex="56"/>
    </mx:FormItem>
  </mx:Form>
</mx:Accordion>
```

For TabNavigator components, use the `getTabAt()` method instead of `getHeaderAt()` to obtain the child component tabs. To correctly set the `tabIndex` on the title text of an MX Panel component, the component must be extended because the titleBar child component is protected and cannot be directly accessed from outside of the MX Panel component.

The MX Text and Label controls are also compound components; they both contain a textField. Unfortunately, the reading order cannot be set correctly for MX Text and Label controls when used with the Flex 4 SDK. This issue does not occur in the Flex 3 SDK, nor does it occur when using the Spark equivalent components with the Flex 4 SDK. Adobe recommends using Spark components instead of MX components whenever an equivalent exists. The reading order of Spark RichText and Label components is controlled by simply setting the `tabIndex` property.

## Skipping repetitive components

Repetitive components that appear on every screen in a Flex application can make the application difficult to use by individuals with disabilities. Users without disabilities quickly learn to ignore such repetitive components—a global navigation feature, for example—until they need them. In contrast, users of screen reader software must listen to and navigate past them at the beginning of each new screen, and keyboard-only users must tab through them on each screen to reach the new page content. It is a good practice to provide a method for users to avoid these items until needed. There are two commonly used methods for achieving this without changing the visual appearance of the Flex application:

- Change the reading order
- Provide a link, button, or shortcut key to skip past the repetitive content

### Changing the reading order

When repetitive navigation links appear above or to the left of content, change the reading order so that screen readers read the content before the navigation links. Screen reader users will encounter the main page content before the repetitive links and buttons. Keyboard-only users can access the content without tabbing through the links or buttons on each screen. If there are many other keyboard-accessible elements on the screen,

provide a shortcut to the links so that screen reader and keyboard-only users can reach them without excessive tabbing; for example:

```
//Action Script
txtPageHeading.tabIndex = 1;
// additional page content not shown
txtLastPageContent.tabIndex = 90;
lnkChapter1.tabIndex= 100;
lnkChapter2.tabIndex= 101;
lnkChapter3.tabIndex= 102;
lnkChapter4.tabIndex= 103;
lnkChapter5.tabIndex= 104;

//MXML
<s:Text id="txtPageHeading" tabIndex="1" />
<!-- additional page content not shown -->
<s:Text id="txtLastPageContent" tabIndex="90" />
<mx:LinkButton id="lnkChapter1" tabIndex="100" />
<mx:LinkButton id="lnkChapter2" tabIndex="101" />
<mx:LinkButton id="lnkChapter3" tabIndex="102" />
<mx:LinkButton id="lnkChapter4" tabIndex="103" />
<mx:LinkButton id="lnkChapter5" tabIndex="104" />
```

### Providing a skip mechanism

A "Skip Navigation" link, button, or shortcut provides an alternative way for users to bypass repetitive navigation links and skip to the main content. When the skip navigation link or button is activated, the focus is placed on the first component of the main content, so further tabbing and screen reader review continues from that point.

```
lnkSkipToContent.tabIndex = 1;
lnkChapter1.tabIndex = 2;
lnkChapter2.tabIndex = 3;
lnkChapter3.tabIndex = 4;
lnkChapter4.tabIndex = 5;
lnkChapter5.tabIndex = 6;
txtPageHeading.tabIndex = 10;
// additional page content not shown
txtLastPageContent.tabIndex = 100;
...
lnkSkipToContent.addEventListener(MouseEvent.CLICK, skipToContent);
function skipToContent(e:KeyboardEvent): void
{
  txtPageHeading.setFocus();
}
```

## Forms

The ability of users to successfully complete forms is critical in ensuring the effective use of most Flex applications. When form label and structure information is not properly defined, users with disabilities—particularly those with visual impairments—face significant challenges in completing forms. To create accessible forms, developers must properly label fields, group fields logically, provide accessible error and instructive text, and allow sufficient time for users to complete the form.

### Setting component name information - form label considerations

To be properly rendered by assistive technology, each form field used within an application must have a label associated with it. In Flex applications there are three main ways to associate labels with elements, which are described in "Setting component name information" on page 7. Developers must ensure that all form

elements provide a label using one of the techniques described in that section. In addition to simply providing the component name information, however, developers must make any supplementary visual information required to complete the form available in the form labels to ensure that the form is usable across the widest array of assistive technology. Commonly this issue arises when form field constraints are conveyed visually, most often to indicate a required field. For the field to be accessible, this constraint information must also be included in the accessible label of the form field. For example, when a field is required then "(required)" or similar text should be added to the component name of the form field. An asterisk may also be used as long as text indicating what the asterisk means is added to the beginning of the form.

```
// ActionScript
txtSearch.accessibilityName = "*Search Term:";

// MXML
<mx:Text id="txtSearch" accessibilityName="*Search Term:" />
```

When FormItem components are used for required fields, set the `required` property to `true`. Screen readers will announce "required" after the accessible name of these required components.

```
<mx:FormItem label="Search Term:"><mx:TextInput id="txtSearch" required="true"/>
    </mx:FormItem>
```

Developers should ensure that any visible form label text is provided in a location that is aligned with operating system style guides and language-specific reading paradigms. In general, this means providing text fields to the right of check boxes and radio buttons, and to the left or above all other form controls. These guidelines are implemented by default by Flex components, but label placement for some controls can be set explicitly using the `labelPlacement` attribute:

```
// ActionScript
chkGender.labelPlacement = mx.controls.ButtonLabelPlacement.RIGHT;

// MXML
<mx:CheckBox id="chkGender" labelPlacement="{ mx.controls.ButtonLabelPlacement.RIGHT}" />
```

**Note**: For Spark radio buttons and check boxes there is no `labelPlacement` attribute. Instead a custom skin must be created to place the check box or radio button label anywhere other than to the right of the control.

In general, developers should ensure that visible labels are not present in the reading or tab order of the document. While this may seem counterintuitive, it prevents assistive technology from rendering the labels multiple times. As a basic example, consider a TextInput control with the visual label "Name" provided above the text entry field:

```
<s:Label text="Name" />
<s:TextInput width="252" height="21" accessibilityName="Name" />
```

When a typical screen reader encounters this set of elements it will first read the visible Label text, "Name." It will then process the text entry field and read, "Text - Name." The user will hear "Name . . . Text – Name". To prevent the screen reader from announcing the name twice, set the `accessibilityEnabled` field of the Spark Label element to `false`; for example:

```
<s:Label text="Name" accessibilityEnabled="false" />
<s:TextInput width="252" height="21" accessibilityName="Name" />
```

When a FormItem component is used to set the label, setting `accessibilityEnabled` to `false` is not feasible because it will disable accessibility for the text input also. See "Silence form items and form headings" on page 28 for instructions on handling this situation.

## Form element grouping

Related form elements should be grouped together with any information that is required to understand them. As an example, consider two radio buttons labeled "Yes" and "No". The button label text makes no sense unless the buttons are associated with the relevant question. When the radio buttons are grouped with the

question, assistive technology will render the text of the question when each button is selected. This ensures that the action of the form control is clear and that a unique text representation is provided for each form field.

Form element groupings must be implemented *programmatically*; merely using visual indications, such as lines or rectangles, on the form is not sufficient. In Flex, this is done using a MX FormHeading, FormItem, or Panel container. Developers should choose the container that is best suited to the type of components in the group. The FormHeading component displays a title above the grouped controls, which is then automatically associated with the children elements via the container's accessible name. The FormHeading component should be used to group components other than radio buttons and check boxes.

The FormItem component should be used to group radio button and check boxes, because it provides a visual group label to the left of radio buttons and check boxes, which have their own labels positioned to the right. One FormItem component should enclose the entire set of radio buttons or check boxes. When radio buttons are used, developers must also define the `groupName` property and a corresponding RadioButtonGroup control to ensure that the radio buttons work together as a group. The following example illustrates the use of FormHeading and FormItem components to group form elements:

```
// FormHeading Example, the text "Registration" will be prepended to the
     accessible name of all fields within the form
<mx:Form>
  <mx:FormHeading label="Registration" />
  <mx:FormItem id="f1" color="#ff0000" required="true" label="Enter Name :">
    <mx:TextInput id="txt1" />
  </mx:FormItem>
  <mx:FormItem label="Enter Password :">
    <mx:TextInput displayAsPassword="true"/>
  </mx:FormItem>
  <mx:FormItem label="Select Gender:">
    <s:ComboBox id="c1" width="150">
      <s:dataProvider>
        <mx:ArrayList>
          <fx:String>Male</fx:String>
          <fx:String>Female</fx:String>
        </mx:ArrayList>
      </s:dataProvider>
    </s:ComboBox>
  </mx:FormItem>
</mx:Form>
<s:Button label="Submit" />

// FormItem example showing proper radio button grouping using
     FormItem and RadioButtonGroup with the groupName attribute
<mx:FormItem label="Do you want to receive emails?:">
  <s:RadioButtonGroup id="RadioGroup1" />
  <s:RadioButton label="Yes" groupName="RadioGroup1" />
  <s:RadioButton label="No" groupName="RadioGroup1" />
</mx:FormItem>
```

In some cases, it may be impractical to provide a visual grouping for multiple form elements. When this is the case, developers should include the name of the group in each element's `accessibleName` property. Such grouping significantly increases the usability of forms for screen reader users. It may be preferable to state the answer first followed by the question in the radio button's accessible name. This enables the user to quickly answer without having to fully listen to the query for each element. This is particularly helpful when there are many radio buttons in the group and the group name (question) is constant. Users can listen to the full accessible name for the first radio button, and proceed through the remaining buttons quickly to select the appropriate answer.

```
//ActionScript
txtLabelReceiveEmails.text = "Would you like to receive emails?";
rbReceiveEmailsYes.accessibilityName = "Would you like to receive emails: Yes";
rbReceiveEmailsNo.accessibilityName = "Would you like to receive emails: No";
// or written with the answer first
rbReceiveEmailsYes.accessibilityName = "Yes - I would like to receive emails";
rbReceiveEmailsNo.accessibilityName = "No - Don’t send me any emails";

//MXML
<s:Text id="txtLabelReceiveEmails" text="Would you like to receive emails?" />
<s:RadioButton id="rbReceiveEmailsYes" accessibilityName =
     "Would you like to receive emails: Yes" />
<s:RadioButton id="rbReceiveEmailsNo" accessibilityName =
     "Would you like to receive emails: No" />
<!-- or written with the answer first -->
<s:RadioButton id="rbReceiveEmailsYes" accessibilityName =
     "Yes - I would like to receive emails" />
<s:RadioButton id="rbReceiveEmailsNo” accessibilityName =
     "No - Don’t send me any emails" />
```

If a RadioButtonGroup component is not present, Flex will automatically generate one as long as the `groupName` property is set on each Spark RadioButton. For this grouping to occur properly, developers must set the same value for the `groupName` property for each radio button in the group. Grouping radio buttons in this way will ensure that users can use arrow keys to cycle through all radio buttons in a group.

**Note**: To ensure form fields are grouped properly, developers must also set a tab order that flows through the form and any groups logically. Generally, proper tab order within a form follows the implied visual tab order or business logic of the form. The implied visual tab order is the order in which users would expect tab stops to occur, given the visual layout of the form. See “Keyboard accessibility” on page 15 and “Reading order” on page 23 for more details.

The number of radio buttons in a group is not typically indicated automatically by assistive technology for Flex applications. This is because the `groupName` property is not exposed through MSAA and assistive technologies have no way to determine if a set of radio buttons is related (other than by their relative proximity in the reading order). If the user must know the radio button count properly use the form, then it should be appended to the accessible name of the radio buttons; for example:

```
// FormItem example showing proper radio button grouping using FormItem
and RadioButtonGroup with the groupName attribute and with
accessibilityName set to override the label property to include the
ordering of the radio buttons
<mx:FormItem label="Do you want to receive emails?:">
  <s:RadioButtonGroup id="RadioGroup1" />
  <s:RadioButton label="Yes" accessibilityName="Yes (1 of 2)" groupName="RadioGroup1" />
  <s:RadioButton label="No" accessibilityName="No (2 of 2)" groupName="RadioGroup1" />
</mx:FormItem>
```

### Instructive text

Instructive text that applies to the entire form should be placed at the top of the form and at the beginning of the form’s reading order. This instructive text may include any general instructions for filling out the form, as well as any instructions specific to assistive technology. Instructions such as “required fields are marked with an

asterisk" should be included here. Instructive text that is specific to any field in the form should be located immediately before the field in the reading order.

```
//ActionScript
txtInstructions.text = "Please complete all fields and activate
     confirm to review your entries.";
txtInstructions.tabIndex = 10;
// form fields following with higher tabIndex values
txtSSNError.text = "Do not enter dashes";
txtSSNError.tabIndex = 15;
txtSSN.tabIndex = 16;

//MXML
<s:Text id="txtInstructions" text="Please complete all fields and
     activate confirm to review your entries." tabIndex="10" />
<!-- form fields following with higher tabIndex values -->
<s:Text id="txtSSNError" text="Do not enter dashes" tabIndex="15" />
<s:TextInput id="txtSSN" tabIndex="16" />
```

### Error messages

After the form is submitted, any error messages should be provided in keyboard-focusable text at the beginning of the form's tab order. Error text provided elsewhere may not be immediately apparent to users of assistive technology, leading them to resubmit the form with the same errors present. Error messages should include both an indication that the overall form is in error as well as a list of the specific fields in error and reasons for the error state of each field.

Error messages may be displayed in red (or any color that stands out), but they should also start with the word "Error." Color cannot be the only means of indicating an error. See "Color" on page 44 for more details.

**Note**: If possible, when form validation errors are detected after submission, move the keyboard focus automatically to the overview error message positioned at the top of the form. This is an exception to the general best practice of not moving keyboard focus automatically, described in "Focus" on page 39. This approach immediately informs users of screen readers and screen magnifiers that form errors exist, so they will not assume the form has been successfully submitted. Use `setFocus()` to set the focus to the error text; for example:

```
btnSubmit.addEventListener(MouseEvent.CLICK, validateForm);

function validateForm(e: MouseEvent): void
{
 var errorText = new Label();
 // positioning of label not shown
 addChild(errorText);
 errorText.tabIndex = 10;
 errorText.text = "Error: Please provide your last name.";
 errorText.setFocus();  // set focus to the error text
}
```

Extra steps should be taken to prevent form entry errors when the data being entered is part of a legal commitment or financial arrangement. In general for these transactions, an accessible application must provide at least one of three methods for avoiding and mitigating these errors: allow the submission to be reversible, allow the user to confirm the data entered, or check the data for errors before completing the submission.

### Field specific errors

When field-specific error information is provided, the text should include any known suggestions on how to correct the form if providing a suggestion would not invalidate the purpose of the field. The suggestion text may include specific examples or the format of information that is required. These directions should be

provided as part of the accessible name for the field. If standard Flex form validation is used, this text will automatically be appended to the `accessibilityName` property of the form field. If the standard Flex form validation is not used, developers can explicitly set this information in the `accessibleName` property of each form field in error.

**Note**: If field-specific information is provided, it should be provided *in addition* to providing an overall indication of the form error state. This ensures that users of assistive technology will be made immediately aware that the form is in error and will have specific instructions on how to fix each error in the form.

```
// import the form validator class
import mx.validators.Validator;
function validateForm(e:MouseEvent):void
{
  // validate the fields requiring validation on submit
  var validatorArray:Array = Validator.validateAll(a1);
  // if no errors found submit data, otherwise focus the error message (not shown)
}
...
<mx:FormItem label="I agree to the license agreement" required="true">
  <mx:CheckBox id="agreement"/>
</mx:FormItem>
...
<mx:Array id="a1">
  <mx:StringValidator source="{agreement}" required="true" property="selected"
     maxLength="4" requiredFieldError="You must accept the license agreement."
      tooLongError="You must accept the license agreement."/>
<mx:Array>
...
<mx:Button label="Submit" click="validateForm(event)" />
```

## Real-time form validation

Some Flex forms are designed with internal validation checks that respond to user input in real time as the user is completing the form. These applications immediately prompt the user with a visible or audible indication when a validation error—such as entering letters in a field that requires numerical input—is detected. Some applications also move the focus to the next input field when the user has entered data consistent with the rules. Often, real-time feedback can improve accessibility. However, real-time validation raises some accessibility challenges not present in forms that are validated upon submission. See, for example, “Avoid forced focus changes” on page 39 for guidance on moving the focus automatically.

When audio is used to indicate an error in a form, additional on-screen text should be provided for users who cannot hear or do not have access to audio. For example, producing an audible beep when the phone number length exceeds ten characters should be accompanied by explanatory error text:

```
...
lblPhoneError: Label = new Label();
lblPhoneError.text = "The phone number can only be 10 characters";
// visual placement not shown
lblPhoneError.tabIndex = 12;
lblPhoneError.visible = false;
addChild(lblPhoneError);
...
txtPhoneNumber.addEventLisentener(TextEvent.TEXT _ INPUT, checkLenght);
function checkLength(e : TextEvent): void
{
  if (txtPhoneNumber.length > 10)
  {
    // produce beep (code not shown)
```

```
    // place visual indicator that field has been over typed
   lblPhoneError.visible = true;
  }
  else      // hide visual indicator
    lblPhoneError.visible = false;
}
...
```

### Time-limited forms and controls

For timed interactions, developers should ensure that forms do not disappear or expire if a response is not received within a specified amount of time. Users with disabilities, who may be unable to complete the form within a predetermined time, need a mechanism to indicate that they require more time. For time-limited tests and similar applications, this guideline may be disregarded because removing the time limit would invalidate the purpose of the activity. In these scenarios, developers should make the user aware of the time-based constraint and provide contact information for users that may need to seek exceptions to the time constraints imposed by the activity.

On time-limited forms, developers should warn the user when the time limit is about to expire and allow the user to extend or eliminate the time limit. When prompted that time is about to expire, the user must be given at least 20 seconds to extend the time limit with a simple action, such as pressing the space bar. If the user indicates that additional time is needed, `setTimeout()` can be used to add time to a timeout counter; for example:

```
var timeoutID: uint;
var timeoutAlertID: uint;
...
timeoutID = setTimeout(timeIsAboutToExpire, 3600000);
...
function alertHandler(e: CloseEvent): void
{
  if (alert.YES)
  {
    clearTimeout(timeoutID);
    timeoutID = setTimeout(timeIsAboutToExpire, 3600000);  // set new timeout
  }
  else
    closeForm();
}

function timeIsAboutToExpire(): void
{
   // display alert
  timeoutAlertID = setTimeout(closeForm, 20000);
  Alert.yesLabel = "Magenta";
  Alert.noLabel = "Blue";
  alert = Alert.show("Your Session is about to Expire.",Alert.YES|Alert.NO, alertHandler);
}
…
function closeForm(): void
{
  // close the form time has expired
  // close the timeout alert if it has not been closed
  // display an alert telling the user that the session had expired
  alert = Alert.show("Your Session Has Ended.");
  ...
}
```

Dynamically Updating Controls

Forms that are dynamically updated—including forms in which new form fields appear or become enabled based on user input—are a special case for accessibility. New form fields should appear later in the form's reading and tab order by setting the `tabIndex` as described in "Keyboard accessibility" on page 15 and "Reading order" on page 23. In general, developers should conform to the following process to ensure updates are accessible:

- Make all visible updates to the screen.
- Update all accessibility information via ActionScript.
- Make a single call to `flash.accessibility.Accessibility.updateProperties()` to notify any active assistive technologies that the screen data has updated.

**Note**: Do not call the `Accessibility.updateProperties()` method more than once per second. When this function is called assistive technologies update their cached version of the accessibility data, which can be processor intensive. Frequent updates can cause the system to slow down. The best practice is to make all required changes to accessibility data and then make a single call to the `Accessibility.updateProperties()` method. As a reminder, if the convenience accessor methods are used, there is no need to call `updateProperties()`, as it is done automatically. See "Reading and tab order in practice" on page 29 for more details.

## Focus

A component is said to have *focus* when it is the center of user interaction. Focus includes three separate but highly interrelated concepts: visual focus, keyboard focus, and programmatic focus. Visual focus refers to a visual cue—such as a yellow or blue rectangle or cursor—that indicates where the next user interaction will take place. Keyboard focus refers to the location at which an action will be performed based on the next user interaction from the keyboard. Programmatic focus is exposed to assistive technology to indicate where user interaction will take place.

When focus is properly indicated and maintained, the user knows where they are in a program and what their next keystroke will do. When an application does not maintain focus properly, most users will be confused and frustrated, and assistive technology users will likely be unable to use the application at all.

In almost all situations, visual, keyboard, and programmatic focus are set to the same component. The FocusManager class manages the focus for Spark and MX components in response to keyboard and mouse activity. This can be seen when focus is moved to the appropriate element when a user presses Tab or Shift+Tab. In general, as long as developers provide keyboard accessibility and set the tab order for applications properly—procedures documented in "Keyboard accessibility" on page 15 and "Reading order" on page 23—Flex applications will provide proper focus control.

While this automatic control is sufficient for most situations, sometimes developers may need to set focus to an element explicitly. For example, when a form is submitted and an error is detected, focus should be moved directly to the error message provided at the top of the form. Similarly, when a new window appears, focus should be placed on the first interactive form field. In these situations developers need to programmatically set focus. Unexpected focus changes, however, should generally be avoided, as they can confuse and frustrate users who cannot visually perceive the change in focus or who must use the keyboard to return to the active component prior to the focus shift.

### Avoid forced focus changes

When focus is changed programmatically from one element to another based on certain conditions being fulfilled, it is said to be *forced*. For example, consider a phone number entry form that contains three form fields for entering the area code, exchange, and extension. After the user types the third number into the area code field, the application moves the focus automatically to the next field. Because this type of unexpected focus change can be disorienting for users of assistive technology who cannot see the visual indications of the change, forced focus changes are to be avoided.

```
// The following code would cause a forced focus change when the user
activates the up/down arrows in a closed dropdown list or combo box
// this type of event should be avoided unless there is a method to
provide keyboard access without a forced focus change
cbSelectState.addEventListener(Event.CHANGE, shiftFocus);
function shiftFocus(): void
{
   txtZipCode.setFocus();
}
```

Forced focus changes can be confusing to all users, and particularly users of screen readers. When a forced focus change is made, developers should notify the user where focus will be placed. For example, if tabbing out of a set of radio buttons after a selection causes focus to move to a different field based on the selected radio button, this should be indicated in the `accessibilityName` of each radio button; for example:

```
rbtUserAccountTypeAdministrator.accessibilityName = "Administrator -
moves focus to the administrator password field when selected after
navigating away from the radio button group";
rbtgrpUserAccountType.addEventListener(FocusEvent.FOCUS _ OUT, setFocusToField);
function setFocusToField(e: FocusEvent): void
{
  switch (e.target)
  {
    case rbtUserAccountTypeAdministrator:
    {
      txtAdministratorPassword.setFocus();
    }
    ...
  }
}
```

**Note**: A forced focus change is generally acceptable if it is in response to the activation of a form's submit button. The focus should move to the top of the next screen or to an error message at the top of the current one. This will assist keyboard-only users in quickly interacting with the new content and enable screen reader users to easily learn if the submission was successful.

```
btnPreferences.addEventListener(MouseEvent.CLICK, goToCart);
function goToCart(e: MouseEvent): void
{
  // if user is not logged in, open login screen (code not shown) and
  // set focus to the first field on the screen, the username field
  txtUserName.setFocus();
}
```

### Radio buttons and combo boxes

Forced focus changes on radio button groups and combo boxes, in which the focus moves to the next element or another screen once a selection has been made, are also to be avoided.

For example, consider a radio button group with five selections. To select the fourth item using the default keyboard control for the group, the user would tab to the radio button group and press the Down Arrow key three times. However, if a forced focus change is implemented to the `onChange` event handler for the radio button group, pressing the Down Arrow key to move from the first radio button to the second would initiate the forced keyboard focus before the user could select the desired option. While this simple example clearly illustrates a problem, the situation is made significantly more frustrating when a selection change triggers an unintended screen change or a more intensive process, such as a lengthy database search.

To avoid this type of forced focus change, developers should keep the focus on the radio button or combo box until the user explicitly opts to move on, for example by pressing the Tab key. A Submit or Next button may be used to invoke the action based on the selected radio button; for example:

```
// 4 Radio buttons are present
<s:RadioButtonGroup id="group1" />
<s:RadioButton label="Small" id="rbtnPizzaSizeSmall" tabIndex= "2"
     groupName="group1" />
<s:RadioButton label="Medium" id="rbtnPizzaSizeMedium" tabIndex = "3"
     groupName="group1" />
<s:RadioButton label="Large"  id="rbtnPizzaSizeLarge" tabIndex = "4"
     groupName="group1" />
<s:RadioButton label="Extra Large" id="rbtnPizzaSizeExtraLarge"
     tabIndex = "5" groupName="group1" />
<s:Button label="Submit" id="btnSubmit" tabIndex ="6" />
...
//ActionScript
btnSubmit.addEventListener(MouseEvent.CLICK, advancedToNextStep );
function advanceToNextStep(e: MouseEvent): void
{
  // advance to next step not shown
}
```

### Visual focus indication

By default, Flex provides a blue rectangle to indicate the current focus for standard components. In addition, Flash Player provides a yellow rectangle for any native Flash objects interspersed with Flex components. Some developers may change the focus rectangle skin to hide the visual focus, perhaps because the color does not match the application theme. When the visual focus is not visible, however, users will have no idea where the next keyboard action will take place. This makes the application exceptionally difficult to use. With few exceptions, this technique will also render an application inaccessible. All accessible applications must provide a visual indication of focus. While Adobe recommends using the default Flex and Flash focus indications, developers are welcome to provide their own form of visual focus indication. Providing no visual focus within an application, however, is not an option for accessible applications. Similarly, providing a visual focus that is not easy to see may cause difficulties for users with visual impairments.

To modify the visual focus indication for a component, set its `focusSkin` property to the class name of the custom focus skin. The custom class can be defined via MXML or ActionScript. The class must implement `get` and `set` methods for the target property and must have an `updateDisplayList()` method. A combination of the Spark Rect primitive and its `fill` property can be used; for example:

```
// set the focusSkin property to MyFocusSkin class
<s:TextInput width="150" accessibilityName="First Name:" focusSkin="MyFocusSkin" />

// mxml file MyFocusSkin.mxml containing the custom focus skin class
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.events.FlexEvent;

            import spark.components.supportClasses.SkinnableComponent;

            private var _ target:SkinnableComponent;

            public function get target():SkinnableComponent
```

```
            {
                return _target;
            }

            public function set target(value:SkinnableComponent):void
            {
                _target = value;

                if (_target.skin)
                    _target.skin.addEventListener(FlexEvent.UPDATE_COMPLETE,
                        skin_updateCompleteHandler, false, 0, true);
            }

             override protected function updateDisplayList(unscaledWidth:Number,
             unscaledHeight:Number):void
            {
                this.setActualSize(target.width, target.height);
                super.updateDisplayList(target.width, target.height);
            }

            private function skin_updateCompleteHandler(event:Event):void
            {
                invalidateDisplayList();
            }

        ]]>
    </fx:Script>

    <s:Rect id="focusRect" top="-1" left="-1" right="-1" bottom="-1">
        <s:fill>
            <s:SolidColor id="bgFill" color="0xC5C551" />
        </s:fill>
    </s:Rect>
</s:Group>
```

### Setting focus when a window appears

Whenever a new screen opens in a Flex application, developers should ensure that focus is set on the first interactive element or the element that the user would use to first interact with the content. If this initial focus is not set, screen readers may announce nothing at all when the screen changes. Setting the focus on the correct element in the screen ensures that all content is read in the intended order from that location forward. Otherwise, the screen reader may *guess* where the focus is and the user may miss important content. In general, focus should be set to the component with the lowest `tabIndex` property that has `tabEnabled` set to `true`.

```
// when the title windows appears set focus to the first field
txtUserName.setFocus();
```

### Return focus to open applications

When keyboard users leave a Flex application to use another application and then return, the focus should be where it was last left, specifically on the last focused element. By default, Flash Player resets focus to the element with the lowest `tabIndex` property value that is enabled every time the user returns to the Flex application. Assistive technology users will need to tab back to the component that had the focus each time they return to the application. If the Flex application is frequently used with other applications to copy, paste,

review, or enter information, developers should correct this focus issue. The following code illustrates one way of resolving this issue:

```
var focusObject: IFocusManagerComponent;
// attach events for watching for when focus is lost and gained
addEventListener(Event.DEACTIVATE, losingFocus);
addEventListener(Event.ACTIVATE, gainingFocus);
...
// called when focus is leaving Flash Player
function losingFocus(e:Event):void
{
  focusObject = focusManager.getFocus();
}
// called when focus is returning to Flash Player
function gainingFocus(e:Event):void
{
  if (focusObject)
    focusManager.setFocus(focusObject);
  //else
    // set focus to most appropriate element
}
```

### Focus for custom components

Proper focus management, including visual, keyboard, and programmatic focus, may not be provided by default for custom components. If custom components are used, developers must take steps to implement and test for proper keyboard, visual, and programmatic focus. The procedures for providing focus to custom components are not within the scope of this document and developers should consult the Flex API for more information.

### Sending events

When creating an accessible application it may be necessary to explicitly fire an accessibility event. While these events are automatically fired in the accessibility implementation classes for each component, sending events manually is sometimes required. For example, in some cases a component can gain focus before a needed screen redraw event is sent. In this situation, the screen reader may not correctly announce the focused item. To solve this, an event can be sent using the `sendEvent()` method of the `flash.accessibility.Accessibility` class. See "Creating a Custom Accessibility Implementation" (*http://goo.gl/crgjZ*) in the Flex documentation for more information on `sendEvent()`.

The `sendEvent()` method takes three required parameters: a component name, the child id (typically `0` if the change is on the component itself), and the hexadecimal number representing the MSAA event to send. (A list of other available events that can be sent are defined in the winuser.h file supplied with the Windows SDK.) Events are sent through Flash Player and thus not all events will actually be sent to assistive technologies. The `sendEvent()` method can be used to send events signaling focus changes, value changes, name changes, state changes, selection changes, window reordering, and alerts, among others. In the example provided below the `sendEvent()` method is used to indicate that the `dfStartDate` field has received focus and to ensure that this focus is properly noticed by assistive technology.

```
import flash.accessibility.Accessibility;
...
static const EVENT _ OBJECT _ FOCUS:uint = 0x8005;
Accessibility.sendEvent(dfStartDate,0,EVENT _ OBJECT _ FOCUS);
...
```

Using the `sendEvent()` method does change the programmatic accessibility information exported by visual interface elements–such as the current focused object, names, values, roles, and states–but rather fires an event alerting assistive technology that the change occurred. This in turn causes the assistive technology to update its representation of the application to match the current reality of the component's accessibility

properties. Thus, developers should be careful to fire events only when the underlying data reflects that such a change has occurred.

### Manually setting focus

The same methods and properties used to set focus manually are often used by code that forcibly shifts focus. However, the events that trigger each of these scenarios are generally different. For example, mouse click events (including when Enter or Space is pressed on the keyboard) and keyboard events listening for shortcut keys are typically used to set focus manually. The primary difference is that the former events are expected by the user to shift focus while the later events are not expected to shift focus. When a focus shift occurs without being linked to an expected action, it may prevent access to all content in a field or form.

```
// assign the Mouse click event handler (also triggered by assistive
technology and the space key)  to the submit button
// when the address form is completed, a hypothetical credit card entry form is displayed
// focus is set to the first field in that screen, the credit card holder name field
btnSubmit.addEventListener(MouseEvent.CLICK, submitForm);
function submitForm(e:MouseEvent): void
{
  // submit the form (not shown)
  // when the next form appears, set focus to the first text input
  txtCreditCardHolderName.setFocus();
}
```

## Color

The colors that are used in Flex applications affect accessibility for users that are blind or have low vision as well as for users with a color deficiency such as color blindness. Color-related accessibility guidelines focus on two requirements: avoiding the use of color as the sole means of providing information and ensuring sufficient contrast.

### Conveying information with color

Color can be an effective way to draw a user's attention to important information, however, when color alone is relied on to convey information, that information will be inaccessible to users who are color blind. In accessible applications, developers must not use color as the only means of communicating any information. Common situations were color is used alone to convey meaning include:

- Indicating action
- Prompting a response
- Distinguishing visual elements
- Communicating selection
- Communicating errors

This requirement is not intended to discourage the use of color. Rather, the goal is to provide additional, alternative methods of conveying the same information whenever color is used. These alternative methods fall into two categories: those available to users who do not have assistive technology and those available to users of assistive technologies.

Providing alternatives to individuals who are not using assistive technology ensures the application is accessible to users who are color blind. For these users, an on-screen alternative that is positioned close to the color-coded material is sufficient. This alternative should appear automatically and not require the user to

perform a task, such as moving the mouse over the material. The following techniques can be used to provide such alternatives:

- Adding the word "Error" to the beginning of an error message, instead of relying on the red color of the message text to indicate an error
- Labeling buttons "Submit" and "Cancel", instead of using green or red images (or any particular color) to convey their purpose
- Adding larger fonts or boldface to colored text that highlights important content
- Adding asterisks or the word "required" to colored text that indicates required form fields
- Adding variable borders, shading, or other symbols to color-coded diagrams, pie charts, and graphs

To support users of screen readers, developers should also make sure that any information conveyed via color is provided programmatically using accessibility properties or other relevant Flex component properties. For example, color is commonly used to indicate state, such as when inactive elements are grayed out. When a component is not enabled, the `enabled` property should be set to `false` rather than simply disabling the event handlers and changing the color manually. When the `enabled` property is set to `false` for a component, this information will be conveyed to users of screen readers and the focus manager will remove the component from the tab order.

```
//ActionScript setting a component's enabled property false
btnSearch.label = "Search";
btnSearch.enabled = false;

//MXML for setting a component's enabled property to false
<s:Button id="btnSearch" label="Search" enabled="false" />
```

When developers use the `FormItem` class and set the `required` property to `true`, an asterisk will be added to the form field's label and it will appear in red. In addition, the word "required" will be added to the accessible name of the form field automatically. This programmatic approach provides multiple ways of indicating that the field is required, which results in an application that is more accessible than one that uses any one technique by itself.

```
//MXML code for setting a textual equivalent for color
<mx:FormItem label="SSN" required="true">
  <s:TextInput id="txtSSN" />
</mx:FormItem>
```

### Provide sufficient contrast

Users with low vision or color blindness may have difficulty reading Flex elements that provide insufficient contrast between foreground and background colors. Some of these users may not be using assistive technology and many will not be using a screen reader. Applications should be designed with color and contrast requirements in mind from the earliest stages. During that time, designers and developers should consider the color contrast of all elements, including:

- Text
- Form fields and labels
- Images, graphs, and charts
- Other components and controls (such as media play buttons)

Decorative elements, essential logos or trademarks, or inactive elements are not required to meet sufficient color contrast requirements.

### Provide options for color, contrast, and text size

Currently, Flex components do not support the color and contrast settings selected by users in the operating system platform. Additionally, font and zoom size selection within the browser do not automatically translate to larger components in a Flex application running in Flash Player. Depending on the regulatory requirements

of the deployed application, providing color and contrast options may be a requirement. If this is the case, the application must allow the user to select from a wide variety of color and contrast options from within the application or via user preferences. These selections enable users with low vision and users with color deficiencies to use the application effectively. Application options should allow users to choose from a full range of color and contrast settings, including at least two high-contrast options—one with dark text on a light background and one with light text on a dark background—and at least two low-contrast options. These options should be stored so they can be retrieved and applied for each user, instead of reverting to hard-coded defaults whenever the application is restarted.

## Flickering

For many users, flickering or blinking content is distracting and annoying. For users with photosensitive epilepsy, it can be dangerous. Content that flashes, blinks, or flickers within certain frequency ranges can induce seizures in these users. Examples of flickering content include:

- Flashing text
- Rapid changes in background color
- Pulsing lights and strobe effects
- Movie scenes that include gun fire or lightning
- Timers or buttons that refresh frequently

It is best to avoid this type of flickering content whenever possible. Providing users with an option to disable flashing content does not provide adequate protection, because users may not notice the option in time. A seizure may occur faster than a user could disable the flashing.

If flickering or flashing content must be used in the application, ensure user safety by applying a blink rate that is either very slow or very fast. U.S. Government standards prohibit content that flashes at a rate between 2 and 55 times per second. This means that any flickering content must blink fewer than 2 times per second or more than 55 times per second. If the application complies with this flicker rate guideline the content will be safe, regardless of the size, color, or duration of the blinking content.

## Resources

See the following resources for additional information on creating accessible Flex applications:

- The Adobe Accessibility Resource Center at *http://www.adobe.com/accessibility/*
- The Adobe Flex Accessibility site at *http://www.adobe.com/accessibility/products/flex/*
- Using Adobe Flex 4 at *http://help.adobe.com/en_US/flex/using/index.html*
- ActionScript 3.0 Reference for the Adobe Flash Platform Reference at *http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/accessibility/Accessibility.html*
- Surfing the Internet with JAWS and MAGic at *http://www.freedomscientific.com/Training/Surfs-Up/_Surfs_Up_Start_Here.htm* (This is a tutorial on using the JAWS screen reader to access web-based content from the maker of JAWS, Freedom Scientific.)

See the following class references for details on Flex accessibility:

- flash.accessibility.Accessibility at *http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/accessibility/Accessibility.html* — The Accessibility class manages communication with assistive technologies.
- flash.accessibility.AccessibilityProperties at *http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/accessibility/AccessibilityProperties.html* — The AccessibilityProperties class defines the core accessibility properties exported by user interface elements to enable access by assistive technologies.

**Adobe Systems Incorporated**
345 Park Avenue
San Jose, CA 95110-2704
USA
*www.adobe.com*

Adobe, the Adobe logo, Flex, ActionScript, Flash, and Flash Builder are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

© 2011 Adobe Systems Incorporated. All rights reserved. Printed in the USA.

95012374 5/09