Flex 4 ACE Guide

Created by Jonnie Spratley

Creating a User Interface (UI)

Identify and describe the basic UI controls used in a Flex application.

(UI controls include: NumericStepper, TextInput, CheckBox, RadioButton).

Alert:

The Alert control is part of the MX component set. There is no Spark equivalent. All Flex components can call the static show() method of the Alert class to open a modal dialog box that contains a message and an optional title, buttons, and icons.

Button and ToggleButton:

The Button and TogggleButton controls are part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead.The Button control is a commonly used rectangular button. Button controls look like they can be pressed, and have a text label, an icon, or both on their face. You can optionally specify graphic skins for each of several Button states. You can create a normal Button control or a ToggleButton control. A normal Button control stays in its pressed state for as long as the mouse button is down after you select it. A ToggleButton stays in the pressed state until you select it a second time. The ToggleButton control is available in Spark. In MX, the Button control contains a `toggle` property that provides similar functionality.Buttons typically use event listeners to perform an action when the user selects the control. When a user clicks the mouse on a Button control, and the Button control is enabled, it dispatches a `click` event and a `buttonDown` event. A button always dispatches events such as the `mouseMove`, `mouseOver`, `mouseOut`, `rollOver`, `rollOut`, `mouseDown`, and `mouseUp` events whether enabled or disabled.You can use customized graphic skins to customize your buttons to match your application's look and functionality. You can give the Button and ToggleButton controls different skins. The control can change the image skins dynamically.

MX ButtonBar and MX ToggleButtonBar:

The ButtonBar control is part of both the MX and Spark component sets. Spark does not define a separate ToggleButtonBar control. You can use the Spark ButtonBar control to replicate the functionality of the MX ToggleButtonBar control. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead.

CheckBox:

The CheckBox control is part of both the MX and Spark component sets. While you can use the MX CheckBox control in your application, Adobe recommends that you use the Spark CheckBox control instead.The CheckBox control is a commonly used graphical control that can contain a check mark or not. You can use CheckBox controls to gather a set of `true` or `false` values that aren't mutually exclusive. You can add a text label to a CheckBox control and place it to the left, right, top, or bottom. Flex clips the label of a CheckBox control to fit the boundaries of the control. When a user clicks a CheckBox control or its associated text, the CheckBox control changes its state from checked to unchecked, or from unchecked to checked. A CheckBox control can have one of two disabled states, checked or unchecked. By default, a disabled CheckBox control displays a different background and check mark color than an enabled CheckBox control.

ColorPicker:

The ColorPicker control is part of the MX component set. There is no Spark equivalent. The ColorPicker control lets users select a color from a drop-down swatch panel. It initially appears as a preview sample with the selected color. When a user selects the control, a color swatch panel appears. The panel includes a sample of the selected color and a color swatch panel. By default, the swatch panel displays the web-safe colors (216 colors, where each of the three primary colors has a value that is a multiple of 33, such as #CC0066).

DateChooser and DateField:

The DateChooser and DateField controls are part of the MX component set. There is no Spark equivalent. The DateChooser and DateField controls let users select dates from graphical calendars. The DateChooser control user interface is the calendar. The DateField control has a text field that uses a date chooser popup to select the date as a result. The DateField properties are a superset of the DateChooser properties.

Image control:

Adobe Flex supports several image formats, including GIF, JPEG, PNG, SVG, and SWF files. You can import these images into your applications by using the Image control. The Image control is part of the MX component set. There is no Spark equivalent. The Spark component set does include a BitmapImage class. This class, however, is only meant for use in FXG components or skin classes, and not for importing images into a main application. Furthermore, you can only use embedded images with the BitmapImage class; the BitmapImage class does not support runtime loading of images.

Hrule and Vrule:

The HRule and VRule controls are part of the MX component set. There is no Spark equivalent. The HRule (Horizontal Rule) control creates a single horizontal line and the VRule (Vertical Rule) control creates a single vertical line. You typically use these controls to create dividing lines within a container.

Hslider and Vslider:

The HSlider and VSlider controls are part of both the MX and Spark component sets. While you can use the MX HSlider and VSlider controls in your application, Adobe recommends that you use the Spark controls instead.You can use the slider controls to select a value by moving a slider thumb between the end points of the slider track. The current value of the slider is determined by the relative location of the thumb between the end points of the slider. The slider end points correspond to the slider's minimum and maximum values.

LinkBar:

The LinkBar control is part of the MX component set. There is no Spark equivalent. A LinkBar control defines a horizontal or vertical row of LinkButton controls that designate a series of link destinations. You typically use a LinkBar control to control the active child container of a ViewStack container, or to create a standalone set of links.

LinkButton:

The LinkButton control is part of the MX component set. There is no Spark equivalent. The LinkButton control creates a single-line hypertext link that supports an optional icon. You can use a LinkButton control to open a URL in a web browser.

NumericStepper:

The NumericStepper control is part of both the MX and Spark component sets. While you can use the MX NumericStepper control in your application, Adobe recommends that you use the Spark control instead.You can use the NumericStepper control to select a number from an ordered set. The NumericStepper control consists of a single-line input text field and a pair of arrow buttons for stepping through the valid values. You can use the Up Arrow and Down arrow keys to cycle through the values. If the user clicks the up arrow, the value displayed increases by one unit of change. If the user holds down the arrow, the value increases or decreases until the user releases the mouse button. When the user clicks the arrow, it is highlighted to provide feedback to the user. Users can also type a legal value directly into the text field. Although editable ComboBox controls provide similar functionality, NumericStepper controls are sometimes preferred because they do not require a drop-down list that can obscure important data. NumericStepper control arrows always appear to the right of the text field.

PupUpAnchor:

The PopUpAnchor control is part of the Spark component set. There is no MX equivalent. The PopUpAnchor control displays a pop-up component in a layout. It has no visual appearance, but it has dimensions. The PopUpAncho

r control is used in the DropDownList control. The PopUpAnchor displays the pop-up component by adding it to the PopUpManager, and then sizes and positions the pop-up component relative to itself. Because the pop-up component is managed by the PopUpManager, it appears above all other controls.With the PopUpAnchor control, you can create various kinds of popup functionality, such as the following:•Click a button or a hyperlink, and a form to submit feedback pops up•Click a button, and a search field pops upThese first two scenarios each contain three controls: the button or hyperlink control, the PopUpAnchor control, and the component that pops up (the form or search field).•Mouse over the top of an application and a menu drops downIn a calendar tool, double-click an appointment block to open an Edit dialog

PopUpButton:

The PopUpButton control is part of the MX component set. There is no Spark equivalent. The PopUpButton control consists of two horizontal buttons: a main button, and a smaller button called the pop-up button, which only has an icon. The main button is a Button control. The pop-up button, when clicked, opens a second control called the pop-up control. Clicking anywhere outside the PopUpButton control, or in the pop-up control, closes the pop-up controlThe PopUpButton control adds a flexible pop-up control interface to a Button control. One common use for the PopUpButton control is to have the pop-up button open a List control or a Menu control that changes the function and label of the main button. The PopUpButton control is not limited to displaying menus; it can display any control as the pop-up control. A workflow application that lets users send a document for review, for example, could use a Tree control as a visual indication of departmental structure. The PopUpButton control's pop-up button would display the tree, from which the user could pick the message recipients.The control that pops up does not have to affect the main button's appearance or action; it can have an independent action instead. You could create an undo PopUpButton control, for example, where the main button undoes only the last action, and the pop-up control is a List control that lets users undo multiple actions by selecting them. The PopUpButton control is a subclass of the Button control and inherits all of its properties, styles, events, and methods, with the exception of the `toggle` property and the styles used for a selected button. The control has the following characteristics:

- The `popUp` property specifies the pop-up control (for example, List or Menu).
- The `open()` and `close()` methods lets you open and close the pop-up control programmatically, rather than by using the pop-up button.
- The `open` and `close` events are dispatched when the pop-up control opens and closes.
- You use the `popUpSkin` and `arrowButtonWidth` style properties to define the PopUpButton control's appearance.

For detailed descriptions, see PopUpButton in *ActionScript 3.0 Reference for the Adobe Flash Platform*.

ProgressBar:

The ProgressBar control is part of the MX component set. There is no Spark equivalent. The ProgressBar control provides a visual representation of the progress of a task over time. There are two types of ProgressBar controls: determinate and indeterminate. A *determinate* ProgressBar control is a linear representation of the progress of a task over time. You can use this when the user is required to wait for an extended period of time, and the scope of the task is known. An *indeterminate* ProgressBar control represents time-based processes for which the scope is not yet known. As soon as you can determine the scope, you should use a determinate ProgressBar control.The following example shows both types of ProgressBar controls:***Top.*** *Determinate ProgressBar control* ***Bottom.*** *Indeterminate ProgressBar control* Use the ProgressBar control when the user is required to wait for completion of a process over an extended period of time. You can attach the ProgressBar control to any kind of loading content. A label can display the extent of loaded contents when enabled.

RadioButton:

The RadioButton and RadioButtonGroup controls are part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead.The RadioButton control is a single choice in a set of mutually exclusive choices. A RadioButton group is composed of two or more RadioButton controls with the same group name. Only one member of the group can be selected at any given time. Selecting an deselected group member deselects the currently selected RadioButton control in the group. While grouping RadioButton instances in a RadioButtonGroup is optional, a group lets you do things like set a single event handler on a group of buttons, rather than on each individual button. The RadioButtonGroup tag goes in the <fx:Declarations> tag.

HscrollBar and VscrollBar:

The HScrollBar and VScrollBar controls are part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead.The VScrollBar (vertical ScrollBar) control and HScrollBar (horizontal ScrollBar) controls let the user control the portion of data that is displayed when there is too much data to fit in the display area. Although you can use the VScrollBar control and HScrollBar control as stand-alone controls, they are usually combined with other components as part of a custom component to provide scrolling functionality. Scrollbar controls consists of four parts: two arrow buttons, a track, and a thumb. The position of the thumb and display of the buttons depends on the current state of the control.

The width of the control is equal to the largest width of its subcomponents (arrow buttons, track, and thumb). Every subcomponent is centered in the scroll bar.

The ScrollBarBase control uses four parameters to calculate its display state:•

- Minimum range value
- Maximum range value
- Current position value; must be within the minimum and maximum range values
- Viewport size; represents the number of items in the range that can be displayed at once and must be equal to or less than the range

For more information on using these controls with Spark containers, see “Scrolling Spark containers” on page 336.

Scroller:

The Scroller control is part of the Spark component set. There is no MX equivalent. The Scroller control contains a pair of scroll bars and a viewport. A viewport displays a rectangular subset of the area of a component, rather than displaying the entire component. You can use the Scroller control to make any container that implements the IViewport interface, such as Group, scrollable.The scroll bars control the viewport's vertical and horizontal scroll position. They reflect the viewport's actual size and content size. Scroll bars are displayed according to the Scroller's vertical and horizontal scroll policy properties. By default, the policy is set to "auto". This value indicates that scroll bars are displayed when the content within a viewport is larger than the actual size of the viewport.

For more information on using this control with Spark containers, see “Scrolling Spark containers” on page 336.

Spinner:

The Spinner control is part of the Spark component set. There is no MX equivalent. The Spinner control lets users step through an allowed set of values and select a value by clicking up or down buttons. It is a base class for the NumericStepper control. You could use Spinner to create controls with a different input or display than the standard text input field used in NumericStepper. Another possibility is to use a Spinner control to control tabs or a menu by assigning different values to tabs or menu items.

SWFLoader:

The SWFLoader control is part of the MX component set. There is no Spark equivalent of this control. The SWFLoader control lets you load one Flex application into another Flex application as a SWF file. It has properties that let you scale its contents. It can also resize itself to fit the size of its contents. By default, content is scaled to fit the size of the SWFLoader control. The SWFLoader control can also load content on demand programmatically, and monitor the progress of a load operation.When loading an applications into a main application, you should be aware of the following factors:**Versioning** SWF files produced with earlier versions of Flex or ActionScript may not work properly when loaded with the SWFLoader control. You can use the loadForCompatibility property of the SWFLoader control to ensure that applications loaded into a main application works, even if the applications were compiled with a different version of the compiler.

Security When loading applications, especially ones that were created by a third-party, you should consider loading them into their own SecurityDomain. While this places additional limitations on the level of interoperability between the main application and the application, it ensures that the content is safe from attack.For more information about creating and loading applications, see "Developing and loading sub-applications" on page 178.The SWFLoader control also lets you load the contents of a GIF, JPEG, PNG, SVG, or SWF file into your application, where the SWF file does not contain a Flex application, or a ByteArray representing a SWF, GIF, JPEG, or PNG.
For more information on the Image control, see "Image control" on page 613.

For more information on using the SWFLoader control to load a Flex application, see "Externalizing application classes" on page 654.

Note: *Flex also includes the Image control for loading GIF, JPEG, PNG, SVG, or SWF files. You typically use the Image control for loading static graphic files and SWF files, and use the SWFLoader control for loading Flex applications as SWF files. The Image control is also designed to be used in custom cell renderers and item editors.* A SWFLoader control cannot receive focus. However, content loaded into the SWFLoader control can accept focus and have its own focus interactions.

MX TabBar:

The TabBar control is part of both the MX and Spark component sets. While you can still use the MX control in your application, Adobe recommends that you use the Spark control instead. For information on Spark TabBar, see "Spark ButtonBar and TabBar controls" on page 497.A TabBar control defines a horizontal or vertical row of tabs. The following shows an example of a TabBar control: As with the LinkBar control, you can use a TabBar control to control the active child container of a ViewStack container. The syntax for using a TabBar control to control the active child of a ViewStack container is the same as for a LinkBar control. For an example, see "MX ViewStack navigator container" on page 566. While a TabBar control is similar to a TabNavigator container, it does not have any children.

For example, you use the tabs of a TabNavigator container to select its visible child container. You can use a TabBar control to set the visible contents of a single container to make that container's children visible or invisible based on the selected tab.

MXVideoDisplay:

The VideoDisplay control is part of both the MX and Spark component sets. While you can still use the MX control in your application, Adobe recommends that you use the Spark control instead.
Continue to use the MX VideoDisplay control to work with cue points or stream live video from a local camera using the attachCamera() method of the Camera class.
Flex supports the VideoDisplay control to incorporate streaming media into Flex applications. Flex supports the Flash Video File (FLV) file format with this control.

Spark VideoPlayer:

The VideoPlayer control lets you play progressively downloaded or streaming video. It supports multi-bit rate streaming and live

video when used with a server that supports these features, such as Flash Media Server 3.5 or later.

The VideoPlayer control contains a full UI to let users control playback of video. It contains a play/pause toggle button; a scrub bar to let users seek through video; a volume bar; a timer; and a button to toggle in and out of fullscreen mode. Flex also offers the Spark VideoDisplay control, which plays video without any chrome, skin, or UI. The Spark VideoDisplay has the same methods and properties as the Spark VideoPlayer control. It is useful when you do not want the user to interact with the control.Both VideoPlayer and VideoDisplay support playback of FLV and F4V file formats, as well as MP4-based container formats.

Identify the purpose of UI containers and when to use them.

Spark includes the following containers:

- Group and DataGroup
- SkinnableContainer, SkinnableDataContainer, Panel, TitleWindow, NavigatorContent, BorderContainer, and Application

  Note: The Panel and NavigatorContent classes are subclasses of the SkinnableContainer class. The information below for SkinnableContainer applies also to the Panel and NavigatorContent classes.

For more information on the Application container, see Application containers.

All Spark containers support interchangeable layouts. That means you can set the layout of a container to any of the supported layout types, such as basic, horizontal, vertical, or tiled layout. You can also define a custom layout.

To improve performance and minimize application size, some Spark containers do not support skinning. Use the Group and DataGroup containers to manage child layout. Use SkinnableContainer and SkinnableDataContainer to manage child layout and to support custom skins.

The Group and SkinnableContainer classes can take any visual components as children. Visual components implement the IVisualElement interface, and include subclasses of the UIComponent class and the GraphicElement class.

The UIComponent class is the base class of all Flex components. Therefore, you can use any Flex component as a child of the Group and SkinnableContainer class.

The GraphicElement class is the base class for the Flex drawing classes, such as the Ellipse, Line, and Rect classes. Therefore, you can use subclass of the GraphicElement class as a child of the Group and SkinnableContainer class.

The DataGroup and SkinnableDataContainer classes take as children visual components that implement the IVisualElement interface and are subclasses of DisplayObject. This includes subclasses of the UIComponent class.

However, the DataGroup and SkinnableDataContainer containers are optimized to hold data items. Data items can be simple data items such as String and Number objects, and more complicated data items such as Object and XMLNode objects. Therefore, while these containers can hold visual children, use Group and SkinnableContainer for children that are visual components.

The following table lists the main characteristics of the Spark containers:

	Children	Skinnable	Scrollable	Creation policy	Primary use
Group	IVisualElement	No	As a child of Scroller	All	Lay out visual children.
DataGroup	Data Item IVisualElement and DisplayObject	No	As a child of Scroller	All	Render and lay out data items.
SkinnableContainer	IVisualElement	Yes	By skinning	Selectable	Lay out visual children in a skinnable container.
SkinnableDataContainer	Data item IVisualElement and DisplayObject	Yes	By skinning	Selectable	Render and lay out data items in a skinnable container.
BorderContainer	IVisualElement	No	No	Selectable	Lay out visual children in a nonskinnable container that includes a border.

NavigatorContent	IVisualElement	Yes	By skinning	Inherited from parent container	Subclass of SkinnableContainer that can be used as the child of an MX navigator container.
Panel	IVisualElement	Yes	By skinning	Selectable	Subclass of SkinnableContainer that adds a title bar and other visual elements to the container.
TitleWindow	IVisualElement	Yes	By skinning	Selectable	Subclass of Panel that is optimized for use as a pop-up window.

For information on skinning, see Spark Skinning. For information on creation policy, see About the creation policy.

Change the look and feel of a design by using API styles, style sheets, Spark skins, filters and blends, and visual customizations by using Halo.

About Spark skins

In the Flex 4 skinning model, the skin controls all visual elements of a component, including layout. The new architecture gives developers greater control over what their components look like a structured and tool-friendly way. Previously, MX components that used the Halo theme for their skins defined their look and feel primarily through style properties.

Spark skins can contain multiple elements, such as graphic elements, text, images, and transitions. Skins support states, so that when the state of a component changes, the skin changes as well. Skin states integrate well with transitions so that you can apply effects to one or more parts of the skins without adding much code.

You typically write Spark skin classes in MXML. You do this with MXML graphics tags (or FXG components) to draw the graphic elements, and specify child components (or subcomponents) using MXML or ActionScript.

The base class for Flex 4 skins is the spark.components.supportClasses.Skin class. The default Spark skins are based on the SparkSkin class, which subclasses the Skin class.

In general, you should try to put all visual elements of a component in the skin class. This helps maintain a necessary separation between the model (the logic and declarative structure of the application) and the view (the appearance of the application). Properties that are used by skins (for example, the placement of the thumb in a slider control) should be defined in the component so that they can be shared by more than one skin.

Most skins use the BasicLayout layout scheme within the skin class. This type of layout uses constraints, which means that you specify the distances that each element is from another with properties such as `left`, `right`, `top`, and `bottom`. You can also specify absolute positions such as the x and y coordinates of each element in the skin.

When creating skins, you generally do not subclass existing skin classes. Instead, it is often easier to copy the source of an existing skin class and create another class from that. Use this method especially if you are going to reuse the skin for multiple instances of a component or multiple components. If you want to change the appearance of a single instance of a component, you can use MXML graphics syntax or apply styles inline.

When creating a Spark skin, you can use MXML, ActionScript, FXG, embedded images, or any combination of the above. You do not generally use run-time loaded assets such as images in custom skins.

Applying skins

You usually apply Spark skins to components by using CSS or MXML. With CSS, you use the `skinClass` style property to apply a skin to a component, as the following example shows:

```
s|Button {
    skinClass: ClassReference("com.mycompany.skins.MyButtonSkin");
}
```

When applying skins with MXML, you specify the name of the skin as the value of the component's `skinClass` property, as the following example shows:

```
<s:Button skinClass="com.mycompany.skins.MyButtonSkin" />
```

You can also apply a skin to a component in ActionScript. You call the `setStyle()` method on the target component and specify the value of the `skinClass` style property, as the following example shows:

```
myButton.setStyle("skinClass", Class(MyButtonSkin));
```

Anatomy of a skin class

Custom Spark skins are MXML files that define the logic, graphic elements, subcomponents, states, and other objects that make up a skin for a Spark component.

The structure of Spark skin classes is similar to other custom MXML components. They include the following elements:

- Skin root tag, or a subclass of Skin (required)
- Host component metadata (optional, but recommended)
- States declarations (required if defined on the host component)
- Skin parts (required if defined on the host component)
- Script block (optional)
- Graphic elements and other controls (optional)

In addition to these elements, Spark skins can contain MXML language tags such as Declarations and Library.

Root tags

Skin classes use the Skin class, or a subclass of Skin such as SparkSkin, as their root tag. The root tag contains the namespace declarations for all namespaces used in the skin class. The following commonly appears at the top of each skin class file:

```
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
```

You can set additional properties on the `<s:Skin>` tag, such as `minWidth` or `scaleX`. You can also set style properties such as `color` and `fontWeight`. In addition, you can specify values based on the state of the control on the root tag. For example, `color.down="0xFFFFFF"`. You cannot set the `includeIn` or `excludeFrom` properties on the root tag of the skin class.

If you create a custom theme for your application, or if you do not need support for global Spark styles in your custom skin class, you can use Skin rather than SparkSkin as your custom skin's root tag. The SparkSkin class adds support for the colorization styles (such as `chromeColor` and `symbolColor`) and supports excluding specific skin parts from colorization, or for specifying symbols to colorize.

Host components

Spark skin classes typically specify the host component on them. The host component is the component that uses the skin. By specifying the host component, Spark skins can gain a reference to the component instance that uses the skin by using the `hostComponent` property. The following example defines the Spark Button as the host component:

```
<fx:Metadata>
    [HostComponent("spark.components.Button")]
</fx:Metadata>
```

Adding the `[HostComponent]` metadata is optional, but it lets Flex perform compile-time checking for skin states and required skin parts. Without this metadata, no compile-time checking can be done.

Skin states

Skin states are skin elements that are associated with component states. They are not the same as component states. For example, when a Button is down, the Button's skin displays elements that are associated with the `down` skin state. When the button is up, the button displays elements that are associated with the `up` skin state.

Skins must declare skin states that are defined on the host component. At runtime, the component sets the appropriate state on the skin. Skin states are referenced by using the dot-notation syntax, *`property.state`* (for example, `alpha.down` defines the value of the `alpha` property in the `down` state).

The following example shows the skin states for the Spark Button skin:

```
 <s:states>
    <s:State name="up" />
    <s:State name="over" />
    <s:State name="down" />
    <s:State name="disabled" />
</s:states>
```

Skin parts

Skin parts are components defined in the skin class and the host component. They often provide a way for a host component to push data into the skin. The component also uses skin parts to hook up behaviors.

Spark container skins include a content group that defines the group where the content children are pushed into and laid out in. This element has an ID of `contentGroup`. All skinnable containers have a `contentGroup`. The content group is a static skin part.

Layouts

Both the Skin and SparkSkin classes use BasicLayout as their default layout scheme. This is the equivalent of having the following defined in the skin class:

```
<s:layout>
    <s:BasicLayout/>
</s:layout>
```

The layout scheme is important when there is more than one graphical element or subcomponent used in the skin. BasicLayout relies on constraints and/or absolute positioning to determine where to place components.

Subcomponents

The Spark skin classes typically include graphic elements and other components that make up the appearance of the skin.

Script blocks

Optionally, the Spark skin class can include a Script block for skin-specific logic.

Most Spark skins have a special `<fx:Script>` block at the top of the skin class. This block typically defines style properties that the skin class respects, including the exclusions that the skin uses. The tag includes a special attribute, `fb:purpose="styling"`:

```
<fx:Script fb:purpose="styling">
```

This attribute is used by Flash Builder. When you create a copy of a skin class in Flash Builder, you can opt to make the skin styleable. If you choose to make it styleable, Flash Builder includes this section of the skin class. If you choose not to make the skin styleable, Flash Builder excludes this section.

Language tags

Like any MXML-based class, you can use the Library tag inside the root tag to declare repeatable element definitions. If you want to use non-visual objects in your skin class, you must wrap them in a Declarations tag.

Versions of included skin classes

While the new Spark skinning architecture makes creating your own skins easy, Flex 4 includes several sets of skins.

The following table describes the skinning packages that ship with Flex 4:

Package	Description
spark.skins.spark.*	Default skins for Spark components.
spark.skins.wireframe.*	A simplified theme for developing applications with a "prototype" look to them. To use wireframe skins, you can apply the wireframe theme or apply the skins on a per-component basis. For information about applying themes, see Using themes.
mx.skins.halo.*	MX skins available for MX components that do not conform to the Spark skinning architecture. You can use these skins in your application instead of the Spark skins by overriding the styles, loading the Halo theme, or by setting the `compatibility-version` compiler option to 3 when compiling your application. For information about these skins, see Skinning MX components.
mx.skins.spark.*	The default skins for MX components when using the default Spark theme. These skins are used by the MX components in Flex 4 applications. These skins give the MX components a similar appearance to the Spark components in Flex 4 applications.
mx.skins.wireframe.*	Wireframe skins for MX components.

Skins typically follow the naming convention *componentName*Skin.mxml. In the *ActionScript 3.0 Reference for the Adobe Flash Platform*, most skins have several versions. For example, there are four classes named ButtonSkin. The default skins for the Spark components are in the spark.skins.spark.* package.

Flex 4 also ships with several themes that use some of the skinning packages. For more information, see About the included theme files.

Skinning contract

The *skinning contract* between a skin class and a component class defines the rules that each member must follow so that they can communicate with one another.

The skin class must declare skin states and define the appearance of skin parts. Skin classes also usually specify the host component, and sometimes bind to data defined on the host component.

The component class must identify skin states and skin parts with metadata. If the skin class binds to data on the host component, the host component must define that data.

The following table shows these rules of the skinning contract:

	Skin Class	Host Component

Host component	`<fx:Metadata>`
	`  [HostComponent("spark.components.Button")]`
	`</fx:Metadata>`
		n/a

Skin states	`<s:states>`
	`  <s:State name="up"/>`
	`</s:states>`
		`[SkinState("up")];`
		`public class Button {`
		`...`
		`}`

Skin parts	`<s:Button id="upButton"/>`	`[SkinPart(required="false")]` `public var:upButton Button;`
Data	`text="{hostComponent.title}"`	`[Bindable]` `public var title:String;`

The compiler validates the `[HostComponent]`, `[SkinPart]`, and `[SkinState]` metadata (as long as the `[HostComponent]` metadata is defined on the skin). This means that skin states and skin parts that are identified on the host component must be declared in the skin class.

For each `[SkinPart]` metadata in the host component, the compiler checks that a public variable or property exists in the skin. For each `[SkinState]` metadata in the host component, the compiler checks that a state exists in the skin. For skins with `[HostComponent]` metadata, the compiler tries to resolve the host component class, so it must be fully qualified.

After you have a valid contract between a component and its skin class, you can apply the skin to the component.

Accessing host components

Spark skins optionally specify a host component. This is not a reference to an instance of a component, but rather, to a component class. You define the host component by using a `[HostComponent]` metadata tag with the following syntax:

```
<fx:Metadata>
    [HostComponent(component_class)]
</fx:Metadata>
```

For example:

```
<fx:Metadata>
    [HostComponent("spark.components.Button")]
</fx:Metadata>
```

When a skin defines this metadata, Flex creates the typed property `hostComponent` on the skin class. You can then use this property to access members of the skin's host component instance from within the skin. For example, in a Button skin, you can access the Button's style properties or its data (such as the label).

You can access public properties of the skin's host component by using the strongly typed `hostComponent` property as a reference to the component.

```
<s:SolidColor color="{hostComponent.someColor as uint}" />
```

This only works with public properties that are declared directly on the host component. You cannot use this to access the host component's private or protected properties.

To access the values of style properties on a host component from within a skin, you are not required to specify the host component.

You can use the `getStyle()` method as the following example shows:

```
<s:SolidColorStroke color="{getStyle('color')}" weight="1"/>
```

You can also access the root application's properties and methods by using the `FlexGlobals.topLevelApplication` property. For more information, see Accessing application properties.

Defining skin states

Each skinnable component has a set of visual skin states. For example, when a Button is down, the Button's skin displays elements that are associated with the `down` skin state. When the button is up, the button displays elements that are associated with the `up` skin state.

To have a valid contract between a skinnable Spark component and its skin, you identify the skin states in the component. Then, define the state's appearance in the component's skin class.

Skin states are declared in the skin class and identify the different states that the component can assume visually. You can define how the visual appearance changes as the skin's state changes in the skin class.

Subclasses inherit the skin states of their parent. For example, the Button class defines the skin states `up`, `down`, `over`, and `disabled`. The ToggleButton class, which is a subclass of Button, declares the `upAndSelected`, `overAndSelected`, `downAndSelected`, and `disabledAndSelected` skin states, in addition to those states defined by the Button control.

Identifying skin states in a component

Part of the contract between a Spark skin and its host component is that the host component must identify the skin states that it supports. To identify a skin state in the component's class, you use the `[SkinState]` metadata tag. This tag has the following syntax:

```
[SkinState("state")]
```

You specify the metadata before the class definition. The following example defines four skin states for the Button control:

```
[SkinState("up")]
[SkinState("over")]
[SkinState("down")]
[SkinState("disabled")]
public class Button extends Component { .. }
```

Defining the skin states in the skin class

The Spark skinning contract requires that you declare supported skin states in the skin class. You can also optionally define the appearance of the skin state in the skin class. Even if you declare a skin state in the skin class, you are not required to define its appearance.

To define a skin state in a skin class:

1. Declare the skin state in a `<states>` tag.
2. Set the value of properties based on the state of the component. This step is optional, but if you don't define the skin state's appearance, then the skin does not change when the component enters that state.

To declare skin states in the skin class, you populate the top-level `<s:states>` tag with an array of State objects. Each State object corresponds to one skin state.

The following example defines four states supported by the Button skin class:

```
<s:Skin ...>
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    ...
</s:Skin>
```

After you declare the skin states in the skin class, you can then define the appearance of the skin states. To do this, you specify the values of the properties based on the skin state by using the dot-notation syntax (*`property_name.state_name`*). To set the value of the `weight` property of a SolidColorStroke object in the `over` state, you specify the value on the `weight.over` property, as the following example shows:

```
<s:SolidColorStroke color="0x000000" weight="1" weight.over="2"/>
```

You can also specify the stateful values of properties by using the `includeIn` and `excludeFrom` properties.

Most commonly, you specify values of style properties based on the state. Flex applies the style based on the current state of the component. The skin is notified when the component's `currentState` property changes, so the skin can update the appearance at the right time.

A common use of this in the Spark skins is to set the `alpha` property of a component when the component is in its `disabled` state. This is often set on the top-level tag in the skin class, as the following example from the ButtonSkin class shows:

```
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21"
    alpha.disabled="0.5">
```

Another example is when a Button label's `alpha` property changes based on other states. The skin sets the value of the labelDisplay's `alpha` property. When the button is in its `up` state, the label has an `alpha` of 1. When the button is in its `over` state, the label has an `alpha` of .25, as the following example shows:

Dynamically change the look of an application by using Spark view states, transitions and effects.

About Effects

An *effect* is a visible or audible change to the target component that occurs over a time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component.

Effects are initiated in response to an event, where the event is often is initiated by a user action, such as a button click. However, you can initiate effects programmatically or in response to events that are not triggered by the user.

You can define multiple effects to play in response to a single event. For example, when the user clicks a Button control, a window becomes visible. As the window becomes visible, it uses effects to move to the lower-left corner of the screen, and resize itself from 100 by 100 pixels to 300 by 300 pixels.

Flex ships with two types of effects: Spark effects and MX effects. Spark effects are designed to work with all Flex components, including MX components, Spark components, and the Flex graphics components. Because Spark effects can be applied to any component, Adobe recommends that you use the Spark effects in your application when possible.

The MX effects are designed to work with the MX components, and in some cases might work with the Spark components. However, for best results, you should use the Spark effects.

Available effects

The following table lists the Spark and MX effects:

Spark Effect	MX Effect	Description

`Animate`
`Animae3D`
`AnimateColor`
	`AnimateProperty`	Animates a numeric property of a component, such as `height`, `width`, `scaleX`, or `scaleY`. You specify the property name, start value, and end value of the property to animate. The effect first sets the property to the start value, and then updates the property value over the duration of the effect until it reaches the end value.
		
		For example, if you want to change the width of a Button control, you can specify width as the property to animate, and starting and ending width values to the effect.

`AnimateFilter`		Animates the properties of a filter applied to the target. This effect does not actually modify the properties of the target. For example, you can use this property to animate a filter, such as a DropShadowFilter, applied to the target.
`Move`	`Move`	Changes the position of a component over a specified time interval. You typically apply this effect to a target in a container that uses absolute positioning, such as a MX Canvas container, or a Spark container that uses BasicLayout. If you apply it to a target in a container that performs automatic layout, the move occurs, but the next time the container updates its layout, it moves the target back to its original position. You can set the container's `autoLayout` property to `false` to disable the move back, but that disables layout for all controls in the container.

`Rotate` `Rotate3D`	`Rotate`	Rotates a component around a specified point. You can specify the coordinates of the center of the rotation, and the starting and ending angles of rotation. You can specify positive or negative values for the angles. Note: To use the rotate effects with MX components that display text, the component must either support the Flash Text Engine or you must use an embedded font with the component, not a device font. For more information, see Styles and themes.
`Scale Scale3D`		Scale a component. You can specify properties to scale the target in the x and y directions.
	`Blur`	Applies a blur visual effect to a component. A Blur effect softens the details of an image. You can produce blurs that range from a softly unfocused look to a Gaussian blur, a hazy appearance like viewing an image through semi-opaque glass. If you apply a Blur effect to a component, you cannot apply a BlurFilter or a second Blur effect to the component. The Blur effect uses the Flash BlurFilter class as part of its implementation. For more information, see flash.filters.BlurFilter in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. You can use the Spark AnimateFilter effect to replace the MX Blur effect.
`CallAction`		Animates the target by calling a function on the target. The effect lets you pass parameters to the function from the effect class.
	`Dissolve`	Modifies the `alpha` property of an overlay to gradually have to target component appear or disappear. If the target object is a container, only the children of the container dissolve. The container borders do not dissolve. Note: To use the MX Dissolve effect with the `creationCompleteEffect` trigger of a DataGrid control, you must define the data provider of the control inline using a child tag of the DataGrid control, or using data binding. This issue is a result of the data provider not being set until the `creationComplete` event is dispatched. Therefore, when the effect starts playing, Flex has not completed the sizing of the DataGrid control.

`Fade` `CrossFade`	`Fade`	Animate the component from transparent to opaque, or from opaque to transparent. If you specify the MX `Fade` effect for the `showEffect` or `hideEffect` trigger, and if you omit the `alphaFrom` and `alphaTo` properties, the effect automatically transitions from 0.0 to the targets' current `alpha` value for a show trigger, and from the targets' current `alpha` value to 0.0 for a hide trigger. Note: To use these effects with MX components that display text, the component must either support the Flash Text Engine or you must use an embedded font with the component, not a device font. For more information, see Styles and themes.
	`Glow`	Applies a glow visual effect to a component. The Glow effect uses the Flash GlowFilter class as part of its implementation. For more information, see the flash.filters.GlowFilter class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. If you apply a Glow effect to a component, you cannot apply a GlowFilter or a second Glow effect to the component. You can use the Spark AnimateFilter effect to replace the MX Glow effect.
	`Iris`	Animates the effect target by expanding or contracting a rectangular mask centered on the target. The effect can either grow the mask from the center of the target to expose the target, or contract it toward the target center to obscure the component. For more information, see Using a MX mask effect.
	`Pause`	Does nothing for a specified period of time. This effect is useful when you need to composite effects. For more information, see Creating composite effects.
`Resize`	`Resize`	Changes the width and height of a component over a specified time interval. When you apply a Resize effect, the layout manager resizes neighboring components based on the size changes to the target component. To run the effect without resizing other components, place the target component in a Canvas container, or a Spark container that uses BasicLayout. When you use the Resize effect with Panel containers, you can hide Panel children to improve performance. For more information, see Improving performance when resizing Panel containers.

	`SoundEffect`	Plays an mp3 audio file. For example, you could play a sound when a user clicks a Button control. This effect lets you repeat the sound, select the source file, and control the volume and pan. You specify the mp3 file using the `source` property. If you have already embedded the mp3 file, using the `Embed` keyword, then you can pass the Class object of the mp3 file to the source property. Otherwise, specify the full URL to the mp3 file. For more information, see Using the MX sound effect.
`Wipe`	`WipeLeft WipeRight WipeUp WipeDown`	Defines a bar Wipe effect. The before or after state of the component must be invisible.
	`Zoom`	Zooms a component in or out from its center point by scaling the component. Note: When you apply a Zoom effect to text rendered using a system font, Flex scales the text between whole point sizes. Although you do not have to use embedded fonts when you apply a Zoom effect to text, the Zoom will appear smoother when you apply it to embedded fonts. For more information, see Styles and themes. You can use the Spark Scale effect to replace the MX Zoom effect.

About View States

In many rich Internet applications, the interface changes based on the task the user is performing. A simple example is an image that changes when the user rolls the mouse over it. More complex examples include user interfaces whose contents change depending on the user's progress through a task, such as changing from a browse view to a detail view. View states let you easily implement such applications.

At its simplest, a view state defines a particular view of a component. For example, a product thumbnail could have two view states; a default view state with minimal information, and a "rich" state with links for more information. The following figure shows two view states for a component:

A base state with minimal information, and a rich state to add the item to a shopping cart

To create a view state, you define a default view state, and then define a set of changes, or *overrides*, that modify the default view state to define the new view state. Each additional view state can modify the default view state by adding or removing child components, by setting style and property values, or by defining state-specific event handlers.

For example, the default view state of the application could be the home page and include a logo, a sidebar, and some welcome content. When the user clicks a button in the sidebar, the application dynamically changes its appearance, meaning its view state, by replacing the main content area with a purchase order form but leaving the logo and sidebar in place.

Two places in your application where you must use view states is when defining skins and item renderers for Spark components. For more information, see Spark Skinning and Item renderer architecture.

Adobe Flash Builder also has built-in support for view states. For more information on using Flash Builder, see Adding View States and Transitions.

Defining a login interface by using view states

One use of view states is to implement a login and registration form. In this example, the default view state prompts the user to log in, and includes a LinkButton control that lets the user register, if necessary, as the following image shows:

Login and registration form

LinkButton control (A)

If the user selects the Need to Register link, the form changes view state to display registration information, as the following image

shows:

If the user selects the Need to Register link, the form changes view state to display registration information.

A.

Modified title of Panel container

B.

New form item

C.

Modified label of Button control

D.

New LinkButton control

Notice the following changes to the default view state to create this view state:

- ☒ The title of the Panel container is set to Register
- ☒ The Form container has a new TextInput control for confirming the password
- ☒ The label of the Button control is set to Register
- ☒ The LinkButton control has been replaced with a new LinkButton control that lets the user change state back to the default view state

When the user clicks the Return to Login link, the view state changes back to the default view state to display the Login form. This change reverses all the changes made when changing to the register view state.

To see the code that creates this example, see Example: Login form application.

Comparing view states to MX navigator containers

View states give you one way to change the appearance of an application or component in response to a user action. You can also use MX navigator containers, such as the Accordion, Tab Navigator, and ViewStack containers when you perform changes that affect several components.

Your choice of using navigator containers or states depends on your application requirements and user-interface design. For example, if you want to use a tabbed interface, use a TabNavigator container. You might decide to use the Accordion container to let the user navigate through a complex form, rather than using view states to perform this action.

When comparing view states to ViewStack containers, one thing to consider is that you cannot easily share components between the different views of a ViewStack container. That means you have to recreate a component each time you change views. For example, if you want to show a search component in all views of a View Stack container, you must define it in each view.

When using view states, you can easily share components across multiple view states by defining the component once, and then including it in each view state. For more information about sharing components among view states, see Controlling when to create added children. For more information on navigator containers, see MX navigator containers.

About Transitions

View states let you vary the content and appearance of an application, typically in response to a user action. To use view states, you define the default view state of an application, and one or more additional view states.

When you change view states, Adobe® Flex® performs all the visual changes to the application at the same time. That means when you resize, move, or in some other way alter the appearance of the application, the application appears to jump from one view state to the next.

Instead, you might want to define a smooth visual change from one view state to the next, in which the change occurs over a period of time. A *transition* is one or more effects grouped together to play when a view state change occurs.

For example, your application defines a form that in its default view state shows only a few fields, but in an expanded view state shows additional fields. Rather than jumping from the base version of the form to the expanded version, you define a transition that uses the Resize effect to expand the form, and then uses the Fade effect to slowly make the new form elements appear on the screen.

When a user changes back to the base version of the form, your transition uses a Fade effect to make the fields of the expanded form disappear, and then uses the Resize effect to slowly shrink the form back to its original size.

By using a transition, you can apply one or more effects to the form, to the fields added to the form, and to fields removed from the form. You can apply the same effects to each form field, or apply different effects. You can also define one set of effects to play when you change state to the expanded form, and a different set of effects to play when changing from the expanded state back to the base state.

Position UI elements by using constraint-based layout.

Using Constraints to control component layout

You can manage a child component's size and position simultaneously by using constraint-based layout, or by using constraint rows and columns. Constraint-based layout lets you anchor the sides or center of a component to positions relative to the viewable region of the component's container. The *viewable region* is the part of the component that is being displayed, and it can contain child controls, text, images, or other contents.

Constraint rows and columns let you subdivide a container into vertical and horizontal constraint regions to control the size and positioning of child components with respect to each other and within the parent container.

Creating a constraint-based layout

You can use constraint-based layout to determine the position and size of the immediate children of any container that supports absolute positioning. With constraint-based layout, you can do the following:

- Anchor one or more edges of a component at a pixel offset from the corresponding edge of its container's viewable region. The anchored child edge stays at the same distance from the parent edge when the container resizes. If you anchor both edges in a dimension, such as top and bottom, the component resizes if the container resizes.
- Anchor the child's horizontal or vertical center (or both) at a pixel offset from the center of the container's viewable region. The child does not resize in the specified dimension unless you also use percentage-based sizing.
- Anchor the baseline of a component at a pixel offset from the top edge of its parent container.

You can specify a constraint-based layout for any Flex framework component (that is, any component that extends the UIComponent class). The following rules specify how to position and size components by using constraint-based layout:

- Place the component in any Spark container that uses BasicLayout, in a MX Canvas container, or in a MX Application or MX Panel container with the `layout` property set to `absolute`.
- Specify the constraints by using the `baseline`, `top`, `bottom`, `left`, `right`, `horizontalCenter`, or `verticalCenter`properties of UIComponent and GraphicElement.

Note: *In previous releases of Flex, the baseline, top, bottom, left, right, horizontalCenter, or verticalCenter properties were implemented as styles. You can still use them as styles in this release.*

The `top`, `bottom`, `left`, and `right` properties specify the distances between the component sides and the corresponding container sides.

The `baseline` constraint specifies the distance between the baseline position of a component and the upper edge of its parent container. Every component calculates its baseline position as the y-coordinate of the baseline of the first line of text of the component. The baseline of a UIComponent object that does not contain any text is calculated as if the UIComponent object contained a UITextField object that uses the component's styles, and the top of the UITextField object coincides with the component's top.
The `horizontalCenter` and `verticalCenter` properties specify distance between the component's center point and the container's center, in the specified direction; a negative number moves the component left or up from the center.
The following example anchors the Form control's left and right sides 20 pixels from its container's sides:

```
<mx:Form id="myForm" left="20" right="20"/>
```

- Do not specify a `top` or `bottom` property with a `verticalCenter` property; the `verticalCenter` value overrides the other properties. Similarly, do not specify a `left` or `right` property with a `horizontalCenter` property.
- A size determined by constraint-based layout overrides any explicit or percentage-based size specifications. If you specify `left` and `right` constraints, for example, the resulting constraint-based width overrides any width set by a`width` or `percentWidth` property.

Precedence rules for constraint-based components

- If you specify a single edge constraint (`left`, `right`, `top`, or `bottom`) without any other sizing or positioning parameter, the component size is the default size and its position is determined by the constraint value. If you specify a size parameter (width or height), the size is determined by that parameter.
- If you specify a pair of constraints (`left-right` or `top-bottom`), the size and position of the component is determined by those constraint values. If you also specify a center constraint (`horizontalCenter` or `verticalCenter`), the size of the component is calculated from the edge constraints and its position is determined by the center constraint value.
- Component size determined by a pair of constraint-based layout properties (`left-right` or `top-bottom`) overrides any explicit or percentage-based size specifications. For example, if you specify both `left` and `right` constraints, the calculated constraint-based width overrides the width set by a `width` or `percentWidth` property.
- Edge constraints override `baseline` constraints.

Example: Using constraint-based layout for a form

The following example code shows how you can use constraint-based layout for a form. In this example, the Form control uses a constraint-based layout to position its top just inside the canvas padding. The form left and right edges are 20 pixels from the outer SkinnableContainer container's left and right edges. The second, inner, SkinnableContainer that contains the buttons uses a constraint-based layout to place itself 20 pixels from the right edge and 10 pixels from the bottom edge of the outer SkinnableContainer container.

Using constraint rows and columns with MX containers

You can subdivide a container that supports absolute positioning into vertical and horizontal constraint regions to control the size and positioning of child components with respect to each other, or with respect to the parent container.

Note: *Constraint rows and columns are only supported by the MX containers. They are not supported by the Spark containers.*

You define the horizontal and vertical constraint regions of a container by using the `constraintRows` and `constraintColumns`properties. These properties contain Arrays of constraint objects that partition the container horizontally (ConstraintColumn objects) and vertically (ConstraintRow objects). ConstraintRow objects are laid out in the order they are defined, from top to bottom in their container; ConstraintColumn objects are laid out from left to right in the order they are defined.

The following example shows a `Canvas` container partitioned into two vertical regions and two horizontal regions. The first constraint column occupies 212 pixels from the leftmost edge of the `Canvas`. The second constraint column occupies 100% of the remaining Canvas width. The rows in this example occupy 80% and 20% of the Canvas container's height from top to bottom, respectively.

Creating constraint rows and columns

Constraint columns and rows have three sizing options: fixed, percent, and content. These options dictate the amount of space that the constraint region occupies in the container. As child components are added to or removed from the parent container, the space allocated to each ConstraintColumn and ConstraintRow instance is computed according to its sizing option.

- *Fixed size* means the space allocated to the constraint region is a fixed pixel size. In the following example, you set the fixed with of a ConstraintColumn instance to 100 pixels:

```
<mx:ConstraintColumn id="col1" width="100"/>
```

  As the parent container grows or shrinks, the ConstraintColumn instance remains 100 pixels wide.

- *Percent size* means that the space allocated to the constraint row or column is calculated as a percentage of the space remaining in the parent container after the space allocated to fixed and content size child objects has been deducted from the available space.

  In the following example, you set the width of a ConstraintColumn instance to 80%:

```
<mx:ConstraintColumn  id="col1" width="80%"/>
```

  As the parent container grows or shrinks, the ConstraintColumn always takes up 80% of the available width.
  A best practice in specifying percent constraints is to ensure that the sum of all percent constraints is less than or equal to 100%. However, if the total value of percent specifications is greater than 100%, the actual allocated percentages are calculated so that the proportional values for all constraints total 100%.

  For example, if the percentages for two constraint objects are specified as 100% and 50%, the values are adjusted to 66.6% and 33.3% (two-thirds for the first value and one-third for the second).

- *Content size* (default) means that the space allocated to the region is dictated by the size of the child objects in that space. As the size of the content changes, so does the size of the region. Content sizing is the default when you do not specify either fixed or percentage sizing parameters.

  In the following example, you specify content size by omitting any explicit width setting:

```
<mx:ConstraintColumn  id="col1"/>
```

  The width of this ConstraintColumn is determined by the width of its largest child. When children span multiple content sized constraint rows or constraint columns, Flex divides the space consumed by the children among the rows and columns.

For the ConstraintColumn class, you can also use the `maxWidth` and `minWidth` properties to limit the width of the column.

For the ConstraintRow class, you can use the `maxHeight` and `minHeight` properties to limit the height of the row. Minimum and

maximum sizes for constraint columns and rows limit how much the constraint regions grow or shrink when you resize their parent containers. If the parent container with a constraint region shrinks to less than the minimum size for that region when you resize the container, scroll bars appear to show clipped content.

Note: *Minimum and maximum limits are only applicable to percentage and content sized constraint regions. For fixed size constraint regions, minimum and maximum values, if specified, are ignored.*

Positioning child components based on constraint rows and constraint columns

Anchor a child component to a constraint row or constraint column by prepending the constraint region's ID to any of the child's constraint parameters. For example, if the ID of a ConstraintColumn is "col1", you can specify a set of child constraints as `left="col1:10"`, `right="col1:30"`, `horizontalCenter="col1:0"`.

If you do not qualify constraint parameters (`left`, `right`, `top`, and `bottom`) a constraint region ID, the component is constrained relative to the edges of its parent container. Components can occupy a single constraint region (row or column) or can span multiple regions.

The following example uses constraint rows and constraint columns to position three Button controls in a Canvas container:

While you can specify any combination of qualified and unqualified constraints, some constraint properties may be overridden. The priority of sizing and positioning constraints are as follows:

1. Center constraint specifications override all other constraint values when determining the position of a control.
2. Next, left edge and top positions are determined.
3. Finally, right edge and bottom positions are calculated to best fit the component.

The following table defines the behavior of constrained components when they are contained in a single constraint region.*Edge 1* is the first specified edge constraint for a child component (`left`, `right`, `top`, or `bottom`). *Edge 2* is the second specified edge constraint of a pair (`left` and `right`, `top` and `bottom`). *Size* is an explicit size for a child component (`width`,`height`). *Center* is the positioning constraint to center the child object (`horizontalCenter` or `verticalCenter`).

Constraint Parameters				**Behavior**	
Edge 1	**Edge 2**	**Size**	**Center**	**Size**	**Position**
x				Default component size	Relative to specified edge constraint
x	x			Calculated from specified edge constraints	Relative to specified edge constraints
x	x	x		Determined from specified edge constraints. Explicit size is overridden	Relative to specified edge constraints
x		x		Specified size	Relative to specified edge
x			x	Default component size	Centered in constraint region; edge constraint is ignored
x	x		x	Calculated from edge constraints	Centered in constraint region
x		x	x	Explicit size	Centered in constraint region; single edge constraint is ignored
			x	Default size of component	Centered in constraint region

Implement application navigation by using navigator containers.

Answer

Customize list-based controls.(Customizing includes: using editors, renderers, label functions)

Answer

Given a layout type, explain the differences and when to use that layout type. (Layout types include: percentage based, constraints based, and custom)

Answer

Create a custom layout. (Including understanding the differences between container and layout)

Answer

Flex system architecture and design

Create and use custom components by using MXML and ActionScript.

Answer

Transfer data within an MXML component by using data bindings. (Including two way

binding)

Create, handle, and dispatch events, including developer created event classes that extend

the Event class.

Identify and describe the implementation and purpose of common software design patterns

that are used in Flex. (Design patterns include: Observer, Command, and Data transfer)

Understand the skinning architecture and the role of the SkinnableContainer class.

Given a method in the component lifecycle explain the purpose of and when to use that

method. (Methods include: CreateChildren, UpdateDisplayList)

Explain how modules are used in the development of a Flex application.

Explain the use case and development workflow for building a custom preloader.

Programming Flex applications with ActionScript

Define and extend an ActionScript class.

Implement an ActionScript interface.

Use access modifiers with classes and class members.

Implement data transfer objects.

Implement accessor methods in ActionScript. (Methods include: explicit and implicit getter

and setter)

Use an ArrayCollection to sort, filter, and provide data.

Implement data validation.

Manipulate XML data by using E4X.

Implement events that function properly in the Flex event framework. (Including:

extends the Event class, call super(), override clone())

Interacting with data sources and servers

Implement real-time messaging by using producers and consumers.

Explain the importance of and implement data paging on data sets.

Understand synchronization and online/offline use cases using data management.

Interact with remote data and services by using Remote Procedure Call (RPC) services.

(Services include: HTTPService, WebService, RemoteObject, URLRequest)

Read, write, and upload local files from the local file system by using Flash Player 10 API. (Including: the use of file filters)

Using Flex in the Adobe Integrated Runtime (AIR)

Given a scenario, compile and export a release build of an AIR application. (Scenarios include: Using Flex Builder, from the command line)

Create, populate, and delete files and directories on a local file system.

Create and customize native windows and menus.

Adding drag-and-drop functionality to and from the desktop.

Install, uninstall, and update an AIR application.

List and describe the AIR security contexts.