

Chapter 1: Creating Spark Skins

[Output: IPH, Print, Web] [Revision Control: Changing]

You create Spark skins by either editing an existing skin class or creating a new skin class for a Spark component. For information about skinning MX components, see [Creating Halo Skins](#).

About Spark skinning

[Output: IPH, Print, Web] [Revision Control: Changing]

In the Spark skinning model, the skin controls all visual elements of a component, including layout.

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. Skins usually specify minimum sizing requirements for the component.

You typically write Spark skin classes in MXML. You do this with MXML graphics tags to draw the graphic elements, and specify child components (or subcomponents) using MXML or ActionScript. Most default Spark skins subclass the `SparkSkin` class. Some Spark skins, such as the `TabBarSkin`, `ButtonBarSkin`, and `ApplicationSkin` classes, subclass the `Skin` class.

When creating custom skins, you should try to encapsulate skinning code. Place code that is used by multiple skins in the component class. Place code that is specific to a particular skin implementation in the skin class.

Most skins use the `BasicLayout` layout scheme within the skin class.

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 FXG syntax or styles inline.

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

Applying skins

[Output: IPH, Print, Web] [Revision Control: Changing]

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.MySkin");
}
```

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:

```
<Button skinClass="com.mycompany.skins.MySkin" />
```

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

[Output: IPH, Print, Web] [Revision Control: Changing]

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:

- SparkSkin or Skin root tag (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

Spark skin classes use either the SparkSkin or Skin class 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:SparkSkin
  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:SparkSkin>` or `<s:Skin>` tags, 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="0xFFFFFFFF"`. You cannot set the `includeIn` or `excludeFrom` properties on the root tag of the skin class.

If you create a fully 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 supports the colorization styles (such as `baseColor` and `symbolColor`) and supports excluding specific skin parts from colorization, or for specifying symbols to colorize. The SparkSkin class also supports focus exclusion.

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.

States

Skin states are skin elements that are associated with component states. For example, when a Button is down, the Button's skin displays elements that are associated with the `down` state. When the button is up, the button displays elements that are associated with the `up` state.

Skins must declare states that are defined on the host component. Skins then define the appearance of states and the behavior of state changes. 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).

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. They can also be a means by which the skin can define behaviors.

Layers

Many of the default skin classes categorize parts of skins in “layers” by specifying a layer in an MXML comment tag. The comments about layers help you to see the logical separation of the skin’s structure. In addition, the comments remind you that skins are built by adding one graphic element on top of the other in the order in which they appear in the skin class. The first element is “covered” by the next, and so on.

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 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. These elements can be groups and there can be any number of graphic elements drawn on the component at runtime.

Script blocks

Optionally, the Spark skin class can include a Script block for skin-specific logic. This is where you can define getters and setters for properties that you want the control to be able to set on your custom skin. For example, the Button control sets a list of elements to exclude from colorization as the `exclusions` property.

Content groups

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.

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

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

Skins typically follow the naming convention `component_nameSkin.mxml`. In the ActionScript Language Reference, most skins have several versions. For example, there are four classes named ButtonSkin. When skinning Spark components, you typically use the skins in the `spark.skins.spark` package.

The following table describes the skinning packages.

Package	Description
spark.skins.spark.*	Default skins for Spark components.
mx.skins.spark.*	The default skins for MX components. 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.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 <code>compatibility-version</code> compiler option to 3 when compiling your application. For information about these skins, see Creating Halo Skins .
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.

Skinning contract

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

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 declares which skin class it uses. It must also 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	Required?
Host component	<code><fx:Metadata> [HostComponent("spark.components.Button")] </fx:Metadata></code>	n/a	No
Skin states	<code><s:states> <s:State name="up"/> </s:states></code>	<code>[SkinStates("up"); public class Button { ... }]</code>	Yes
Skin parts	<code><s:Button id="upButton"/></code>	<code>[SkinPart(required="false")] public var upButton Button;</code>	Yes
Data	<code>text="{hostComponent.title}"</code>	<code>[Bindable] public var title:String;</code>	No

The compiler validates the `[HostComponent]`, `[SkinPart]`, and `[SkinState]` metadata. 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

[[Chunk: No](#)] [[Output: IPH, Print, Web](#)] [[Revision Control: Changing](#)]

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:

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

For example:

```
<Metadata>
    [HostComponent ("spark.components.Button")]
</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.backgroundColor as uint}" />
```

This only works with public properties that are declared directly on the host component (such as the `Application`'s `backgroundColor` property). 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 can use the `getStyle()` method as the following example shows:

```
<s:SolidColorStroke color="{hostComponent.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](#)" on page 20.

Defining skin states

[[Chunk: No](#)] [[Output: IPH, Print, Web](#)] [[Revision Control: Changing](#)]

Skin states are skin elements that are associated with component states. For example, when a `Button` is down, the `Button`'s skin displays elements that are associated with the `down` state. When the button is up, the button displays elements that are associated with the `up` state.

To have a valid contract between a 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.

When a component's state changes, the skin usually changes to reflect this state change. 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.

A component manages its state internally. When a component's state changes, the component puts the skin in the correct state so that the component updates visually. A component tracks its current state by setting the `currentState` property, which exists on every skinnable and non-skinnable component. The component handles setting this property. Unless you add a new state, you will not have to set this property directly.

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

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

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

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

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.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"/>
```

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

A `Button` label's `alpha` property typically changes based on other states, too. 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:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/StatesButtonExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Button label="Alpha Changes" skinClass="mySkins.MyAlphaButtonSkin"/>
</s:Application>
```

The skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\MyAlphaButtonSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">

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

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    alpha.up="1"
    alpha.down=".1"
    alpha.over=".25"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
</s:SparkSkin>

```

You can set any property on a component based on the state. You are not limited to style properties. For example, you can change the label of a Button control in its skin based on its state, as the following example shows:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/StyleWatcherExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Button id="myButton" label="Basic Button" skinClass="mySkins.ChangeLabelBasedOnState"/>
</s:Application>

```

The custom skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\ChangeLabelBasedOnState.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    text.up="UP" text.over="OVER" text.down="DOWN"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
</s:SparkSkin>

```

If you identify a skin state on the component, but do not declare it on the skin class, Flex throws a compiler error. You are not required to define the appearance of the skin state in the skin class, but you are required to declare it in the skin class.

If you try to set the value of a style property in a skin class on a state that is not identified by the host component's class, Flex throws a compiler error. This error checking helps to enforce the contract between the component and the skin.

In addition to conditionalizing the values of properties, you can also include or exclude graphic elements from the skin based on the component's state. To do this, you use the `includeIn` and `excludeFrom` properties to define which states a graphic element applies to.

The following example displays a partially transparent gray box when the Button control is in the `down` state:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/IncludeButtonExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Button label="Click Me In the Button Class" skinClass="mySkins.MyIncludeButtonSkin"
color="0xCCC333"/>
</s:Application>

```

The custom skin class for this example is as follows:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\MyIncludeButtonSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
  <!-- Highlight stroke (down state only) -->
  <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" includeIn="down">
    <s:fill>
      <s:SolidColor color="0x000000" alpha="0.25"/>
    </s:fill>
  </s:Rect>
</s:SparkSkin>
```

When you exclude a graphic element with the `excludeFrom` property, Flex removes it from its parent's child list. The element can no longer be referenced.

If you do not specify either the `includeIn` or `excludeFrom` property for a graphic element in a skin class, the graphic element is used in all states of the control by default.

You can specify multiple states to be included or excluded by using a comma-separated list of states for the `includeIn` or `excludeFrom` properties. The following example excludes the `Rect` graphic element from the host component's `over` and `down` states:

```
<s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" excludeFrom="over, down">
```

The `excludeFrom` and `includeIn` properties can be set only in MXML. You cannot set the values of these properties in ActionScript.

All classes support the `includeIn` and `excludeFrom` properties when they are used in the skin class, as long as they are a visual child of a visual parent. This means that you cannot set these properties on the root tag of the skin file, nor can you set them on scalar properties. For example, the following shows a valid and an invalid use of the `includeIn` property:

```
<s:states>
  <s:State name="StateA" />
</s:states>

<!-- This is a valid use of the includeIn property: -->
<s:Button>
  <s:label.StateA>
    <fx:String>My Label</fx:String>
  </s:label.StateA>
</s:Button>

<!-- This is an invalid use of the includeIn property: -->
<s:Button>
  <s:label>
    <fx:String includeIn="StateA">My Label</fx:String>
  </s:label>
</s:Button>
```

If you exclude some graphic elements from a skin, the component sometimes resizes itself or its appearance fluctuates as it cycles through its states. For example, if you set the value of the `excludeFrom` property to `down` on a button skin's `labelDisplay`, the label disappears when the user clicks the button. As a result, the button resizes itself in the `down` state. This behavior is expected and is shown in the following example:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/HideLabelOnDownExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Button id="myButton"
    skinClass="mySkins.HideLabelOnDownSkin"
    label="This Is A Long Button Label"/>
</s:Application>
```

The custom skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\HideLabelOnDownSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect
    left="0" right="0"
    top="0" bottom="0"
    width="69" height="20"
    radiusX="2" radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    excludeFrom="down"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
</s:SparkSkin>

```

The button resizes because its size is dynamically determined by the contents of the label by default. Excluding the label results in the label being removed from its parent.

To prevent the Button from resizing itself, you can set the `alpha.down` property to 0 or the `visible.down` property to `false` rather than excluding the component altogether. The label then disappears but the button does not resize itself because the label is not removed from its parent, as the following example shows:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/HideLabelOnDownExample2.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Button id="myButton"
    skinClass="mySkins.HideLabelOnDownSkin2"
    label="This Is A Long Button Label"/>
</s:Application>

```

The custom skin class for this example is as follows:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\HideLabelOnDownSkin2.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect
    left="0" right="0"
    top="0" bottom="0"
    width="69" height="20"
    radiusX="2" radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    visible.down="false"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
</s:SparkSkin>

```

Applying skin classes to a component

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

After you have defined a skin, you can apply it to a component in one of the following ways:

- CSS
- MXML
- ActionScript

To associate a skin with a component in CSS, you set the value of the `skinClass` property in the style sheet. You can use either type or class selectors to apply a skin class to a component.

The following example applies the `mySkins.NewPanelSkin` class with all `Panel` containers by using a type selector:

```

s|Panel {
  skinClass:ClassReference("mySkins.NewPanelSkin");
}

```

The following example associates the `mySkins.CustomButtonSkin` class with all components that use the `myButtonStyle` class selector:

```
.myButtonStyle {
    skinClass:ClassReference("mySkins.CustomButtonSkin");
}
```

To associate a skin with a component in MXML, you set the value of the `skinClass` property inline. The following example applies the `mySkins.NewPanelSkin` class to this instance of the `Panel` container:

```
<s:Panel skinClass="mySkins.NewPanelSkin"/>
```

The advantage to applying skins with CSS is that with CSS you can use type and class selectors to apply the skin to all components of a particular type (such as all `Buttons`) or all classes that are in a particular class (such as all components with the style name “`myButtonStyle`”).

CSS supports inheritance. If you apply a skin class to a `Button` control in the `Button` type selector, the skin applies to all `Button` controls and subclasses of `Button` controls, such as the `ToggleButton` control. This can have unexpected results in the case of subcomponents. If you apply a new skin to all `Label` controls, components that have `Label` subcomponents will also use that skin. As a result, you should generally use class selectors for applying skins to basic components.

Setting a skin class in MXML lets you only apply the skin to the instance of the component.

In MXML, there is some simple inheritance based on the location of the component in the display list. For example, if you create a skin for a container, child containers also use that skin. But in general, use CSS to define states and manage their inheritance.

You can also apply a skin class to a component in ActionScript. Call the `setStyle()` method on the component, and set the value of the `skinClass` style property to the skin class. You must cast the skin class as a `Class` in the `setStyle()` method.

The following example applies the custom skin class to the `Button` when you click the `Button`:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/SimpleLoadExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        import mySkins.*;

        private function loadSkinClickHandler():void {
            myButton.setStyle("skinClass", Class(MyButtonSkin));
        }
    </fx:Script>
    <s:Button id="myButton"
        label="Click Me"
        color="0xFFFFFF"
        click="loadSkinClickHandler()"/>
</s:Application>
```

The custom skin class for this example is as follows:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\MyButtonSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
</s:SparkSkin>
```

Be sure to import the appropriate skin class package before you can use the class in your application.

As with CSS, you can use ActionScript to apply a skin class to all instances of a particular component type. Call the `setStyle()` method on the component's style declaration. The following example applies the custom Button skin to all buttons in the application:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ASStyleSelectorLoadExample.mxml -->
<mx:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    import mySkins.*;

    private function loadSkinClickHandler():void {

mx.styles.StyleManager.getStyleDeclaration("spark.controls.Button").setStyle("skinClass",
Class(MyButtonSkin));
    }
  </fx:Script>
  <s:Button id="myButton1" label="Click Me" color="0xFFFFFF" click="loadSkinClickHandler()" />
  <s:Button id="myButton2" label="Click Me" color="0xFFFFFF" click="loadSkinClickHandler()" />
  <s:Button id="myButton3" label="Click Me" color="0xFFFFFF" click="loadSkinClickHandler()" />
</mx:Application>
```

Skin parts

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

Some components are composed of individual skin parts. For example, a `NumericStepper` contains an up button, a down button, and text. These parts are declared by the component. The skin defines their appearance. Parts defined by the component can be optional or required in the skin.

Skin parts are an important part of the skinning contract. They let the component instance interact with the skin and can be used to push data down into a skin or to define behaviors.

Skin parts do not have to be top-level tags in the Spark skin class. They can be anywhere in the MXML file.

To declare a skin part on a component, you use the `[SkinPart]` metadata. For example, the `Button` class defines a skin part for the label in the `ButtonBase` class:

```
[SkinPart(required="false")]
public var labelDisplay:TextBase;
```

The `ButtonSkin` class defines a `Label` component as the `labelDisplay` part:

```
<s:Label id="labelDisplay"
  textAlign="center"
  verticalAlign="middle"
  maxDisplayedLines="1"
  horizontalCenter="0" verticalCenter="1"
  left="10" right="10" top="2" bottom="2">
</s:Label>
```

In this example, the `labelDisplay` skin part is technically not required (`required="false"`), but without it, Flex would not draw a label on the `Button` control. The default value for the `required` property is `false`. The contract between the `Button` class and its skin dictates that when you set the `label` property on a button, the value is pushed down into the skin and modifies the value of the `labelDisplay` skin part's text.

Parts are identified by their `id` attributes. The `id` attribute of the skin part in the skin class must match the property name of the skin part in the host component. This helps enforce the contract between the skin and the host component. For example, if you have the following declaration in your component:

```
[SkinPart(required="true")]
public var textInput:TextInput;
```

The skin class sets the `id` attribute of a `TextInput` instance to “textInput”, as the following example shows:

```
<s:TextInput id="textInput" ... />
```

There are two primary types of skin parts: static and dynamic. *Static parts* are instantiated automatically by the skin. There can be only one instance of a static part. For example, the `ScrollBar` class has four static parts: `track`, `thumb`, `upArrow`, and `downArrow`. The `Button` class has one static part: `label`.

Dynamic parts are instantiated when needed. There can be more than one instance of a dynamic part. Dynamic parts are defined in the `Declarations` block of a skin so that one or more instances of the part can be instantiated and used by the skin. The `Slider` class’s `DataTip`, for example, is a dynamic skin part. In the `VSliderSkin` and `HSliderSkin`, the `dataTip` skin part is defined in a `Declarations` block.

In your host component, you specify the type for a dynamic skin part as an `IFactory`. For example, in the `Slider` class, the `dataTip` is defined as:

```
[SkinPart(required="false", type="mx.core.IDataRenderer")]
public var dataTip:IFactory;
```

The `[SkinPart]` metadata is compiled into static information that is defined on each class. This static construct includes the skin parts defined on that class as well as any defined on any base classes.

The component defines a protected getter, `skinParts`, that gets overridden by subclasses and returns all the skin parts for this component. The compiler automatically does this work for you.

At runtime, when a component skin is loaded, the `Component` base class looks at the `skinParts` getter to find all the skin parts, assigns parts that were found, and throws a runtime error if any required parts are not defined by the skin. Deferred parts are found (because they are defined properties of the skin), but have a value of `null` when the component is first instantiated.

Documenting skin states and skin parts

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

The `ASDoc` utility supports documenting the `[SkinStates]` and `[SkinPart]` metadata tags.

You can document the `[SkinStates]` metadata tag by adding comments above the tag in the component’s class file, as the following example shows:

```
/* Up state of the button. */
[SkinState("up")]
```

To document a `[SkinPart]` metadata tag, the `ASDoc` utility uses the description of the variable from the component’s class file. You do not add a comment on the actual metadata tag, as the following example shows:

```
[SkinPart(required="false")]
/*
 * A skin part that defines the label of the button.
 */
public var labelDisplay:TextGraphicElement;
```

The `ASDoc` utility adds skin state documentation to the “Skin States” section of the class’s documentation. It adds skin part documentation to the “Skin Parts” section of the class’s documentation.

For more information about the `ASDoc` utility, see [Using ASDoc](#).

Creating skins from source files

[Output: IPH, Print, Web] [Revision Control: Changing]

When creating a custom Spark skin, the easiest way to get started is to use the source of an existing skin of a similar type as your base class. You can use the default Spark skins as a good starting point for creating skins, or the skins in the Wireframe theme. The Wireframe theme defines a set of simple skins that provide a “prototype” look to the application. They are lighter weight than the default skins but have less functionality.

The default Spark skins are more heavyweight than the Wireframe skins, but they let you maintain the default Spark look and feel. For example, if you are creating a custom Button skin, you can open the `spark.skins.spark.ButtonSkin.mxml` file. Save this file in a different location with a different name. You can then edit the contents of this file for your new custom skin.

The default skins have names that are similar to the component class name (for example, `ToggleButton` uses the `ToggleButtonSkin` class). If you are unsure of the skin class name is used by a component, look in the `defaults.css` file. This style sheet defines all the default skin classes for the components that have them.

In Flash Builder, you can create a new skin based on an existing skin in Design Mode. In the Properties view, click the Pencil icon for Skin and select `Create New Copy of Skin`. Flash Builder creates a file that implements a Skin class for the selected component. Flash Builder modifies the application file by specifying the new skin class as the `skinClass` property of the current component. Flash Builder also opens the generated Skin class file in Design mode of the MXML editor. For more information on using Flash Builder to design custom skins, see [Generating and editing skins for Spark components](#).

Another way to create a spark skin is to generate its graphical elements from a tool that outputs FXG. FXG is a graphics interchange format that defines graphic elements. You can create a graphic in a tool such as Adobe Illustrator that can then export the instructions for drawing that graphic in the FXG format. You copy the FXG output directly into your Spark skin class and with very little modification, use it to define the appearance of your skins. You can also use FXG output as a custom component in your skin class. This method of using FXG is more efficient because the compiler optimizes FXG custom component source code. For more information about using FXG and converting it to MXML graphics tags, see [Using FXG in applications built with Flex](#).

Skinning Spark components

[Output: IPH, Print, Web] [Revision Control: Changing]

All Spark components and subcomponents that are subclasses of `SkinnableComponent` can be reskinned. Common tasks when reskinning Spark components include adding borders, drop shadows, and transitions to a skin class.

Setting minimum sizes of a skin

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

A common task when creating Spark skins is to set minimum sizes of the skin. This causes the Flex layout to not reduce the size of a component below a pre-defined width or height. You do this with the `minWidth` and `minHeight` properties on the skin class's top-level element.

In general, you should not change the values of the minimum height and minimum width properties once they are set.

The following example creates a custom Button skin that has a minimum width of 100 pixels and a minimum height of 100 pixels.

```
<codeblock> Resolved code-reference.
<?xml version="1.0"?>
<!-- SparkSkinning/SquareButtonExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Button id="myButton" skinClass="mySkins.SquareButtonSkin" label="Click Me"/>

</s:Application>
```

The following is the custom skin class used for this example:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="100" minHeight="100"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <fx:Script>
    <![CDATA[
      static private const exclusions:Array = ["labelDisplay"];
      override public function get colorizeExclusions():Array {return exclusions;}
    ]]>
  </fx:Script>

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

  <!-- layer 1: shadow -->
  <s:Rect left="-1" right="-1" top="-1" bottom="-1" radiusX="2" radiusY="2">
    <s:fill>
      <s:LinearGradient rotation="90">
        <s:GradientEntry color="0x000000"
          color.down="0xFFFFFFFF"
          alpha="0.01"
          alpha.down="0" />
        <s:GradientEntry color="0x000000"
          color.down="0xFFFFFFFF"
          alpha="0.07"
          alpha.down="0.5" />
      </s:LinearGradient>
    </s:fill>
  </s:Rect>

  <!-- layer 2: fill -->
```

```

<s:Rect left="1" right="1" top="1" bottom="1" radiusX="2" radiusY="2">
  <s:fill>
    <s:LinearGradient rotation="90">
      <s:GradientEntry color="0xFFFFFFFF"
        color.over="0xBBBDBD"
        color.down="0xAAAAAA"
        alpha="0.85" />
      <s:GradientEntry color="0xD8D8D8"
        color.over="0x9FA0A1"
        color.down="0x929496"
        alpha="0.85" />
    </s:LinearGradient>
  </s:fill>
</s:Rect>

<!-- layer 2: border -->
<s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
  <s:stroke>
    <s:LinearGradientStroke rotation="90" weight="1">
      <s:GradientEntry color="0x000000"
        alpha="0.5625"
        alpha.down="0.6375" />
      <s:GradientEntry color="0x000000"
        alpha="0.75"
        alpha.down="0.85" />
    </s:LinearGradientStroke>
  </s:stroke>
</s:Rect>
<s:Label id="labelDisplay"
  textAlign="center"
  verticalAlign="middle"
  lineBreak="toFit"
  truncation="1"
  horizontalCenter="0" verticalCenter="1"
  left="10" right="10" top="2" bottom="2">
</s:Label>

</s:SparkSkin>

```

Accessing application properties

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

You can access properties of the host component's parent application. This is useful if there are global settings that you want to access, or runtime information that is passed into the application that you want available in the skin class.

To access global variables in a custom skin, you use the `FlexGlobals.topLevelApplication` property. Using this property gives you access to all global variables, including variables that were passed into the application as `flashVars` variables.

The following example accesses a `String` that is a global variable on the application:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/SimpleButtonExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    public var s:String = "Hello World";
  </fx:Script>
  <s:Button skinClass="mySkins.GlobalVariableAccessorSkin"/>
</s:Application>
```

The following is the custom skin class for this example:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\GlobalVariableAccessorSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>
  <fx:Script>
    import mx.core.FlexGlobals;

    [Bindable]
    private var localString:String = FlexGlobals.topLevelApplication.s;
  </fx:Script>

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect
    left="0" right="0"
    top="0" bottom="0"
    width="69" height="20"
    radiusX="2" radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    text="{localString}"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
</s:SparkSkin>
```

Using style properties in custom skins

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

Spark skins expose a limited number of style properties that you can set as styles in the application. For example, you can set the `baseColor` property on all skins that extend `SkinnableComponent`, which includes all component skins. On other skins, such as container skins, you can set the `backgroundColor`, because those skins extend the `SkinnableContainer` class.

You can get the value of style properties on the host component of a custom skin and set values within that skin using the property's value.

The following example sets the color of the border inside the skin to the same value as the `Button`'s `color` style property:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BorderColorButtonExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Button id="myButton"
    label="Basic Button"
    skinClass="mySkins.BorderColorButtonSkin"
    color="green"/>
</s:Application>
```

The following is the skin class for this example:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\BorderColorButtonSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21"
  creationComplete="initSkin()">

  <fx:Script>
    private function initSkin():void {
      outline.color = hostComponent.getStyle('color');
    }
  </fx:Script>

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

```

<!-- Specify one state for each SkinState metadata in the host component's class -->
<s:states>
  <s:State name="up"/>
  <s:State name="over"/>
  <s:State name="down"/>
  <s:State name="disabled"/>
</s:states>
<s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
  <s:stroke>
    <s:SolidColorStroke id="outline" weight="1"/>
  </s:stroke>
</s:Rect>

<s:Label id="labelDisplay"
  horizontalCenter="0" verticalCenter="1"
  left="10" right="10" top="2" bottom="2">
</s:Label>
</s:SparkSkin>

```

You can use a binding expression to set properties in the skin based on values in the host component. The following example binds the value of the stroke's `color` property to the host component's `color` property:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/StyleWatcherExample.mxml -->
<s:Application

  xmlns:fx="http://ns.adobe.com/mxml/2009"

  xmlns:mx="library://ns.adobe.com/flex/halo"

  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Button id="myButton" label="Basic Button" skinClass="mySkins.StyleWatcherSkin"
click="myButton.setStyle('color','red')" color="green"/>
</s:Application>

```

The following is the skin class for this example:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\StyleWatcherSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">

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

  <!-- Specify one state for each SkinState metadata in the host component's class -->
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
    <s:stroke>
      <!-- Match the border color to the button label's color. -->
      <!-- Note that this color is set when the component is first created, but does not
      get updated when you click the button. The color style property is not bindable
      unless you specifically trigger an event. -->
      <s:SolidColorStroke color="{hostComponent.getStyle('color')}" weight="1"/>
    </s:stroke>
  </s:Rect>

  <s:Label id="labelDisplay"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
  </s:Label>
</s:SparkSkin>

```

This example sets the color of the stroke when the application starts. When you click the Button, the `color` property later changes while the application is running because style properties are bindable.

To expose a property as a style, you can also define a getter and setter for the property. You then override the `styleChanged()` method to dispatch the binding change event. In the application, you call the `setStyle()` method on the instance's `skin` property, which sets the value of the style property on the skin.

The following example exposes a custom style property called `borderColor` on the custom skin:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/StyleableBorderSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">

  <fx:Script>
    [Bindable("borderColorChange")]

    public function get borderColor():uint {
      return getStyle("borderColor");
    }

    public function set borderColor(value:uint):void {
      setStyle("borderColor", value);
    }

    override public function styleChanged(styleProp:String):void {
      super.styleChanged(styleProp);

      if (styleProp == "borderColor" || styleProp == null)
        dispatchEvent(new Event("borderColorChange"));
    }
  </fx:Script>
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>
  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>
  <s:Group id="contentGroup" left="0" right="0" top="0" bottom="0">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:stroke>
      <s:SolidColorStroke color="{borderColor}" weight="1"/>
    </s:stroke>
  </s:Rect>
</s:SparkSkin>

```

In the following application, you set the color of the border by using the ColorPicker control. This control uses the `setStyle()` method to change the color of the container's border.

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/StyleableBorderExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" height="200" width="200">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <fx:Script>
    private function setSkinStyles(e:Event):void {
      myContainer.skin.setStyle("borderColor",e.currentTarget.selectedColor);
    }
  </fx:Script>

  <fx:Style>
  @namespace s "library://ns.adobe.com/flex/spark";
  @namespace mx "library://ns.adobe.com/flex/halo";
  </fx:Style>

  <s:SkinnableContainer id="myContainer"
    height="200" width="200"
    skinClass="mySkins.StyleableBorderSkin"
  >
    <mx:ColorPicker id="myColorPicker" change="setSkinStyles(event)"/>
  </s:SkinnableContainer>
</s:Application>

```

Another way expose style properties to the skin is to add code to the `updateDisplayList()` method to manually push style values into the skin's graphics, as the following example shows. In most cases, overriding this method is not necessary because style properties are bindable.

```

<fx:Script>
  override protected function updateDisplayList(unscaledWidth:Number,
  unscaledHeight:Number):void {
    // Push style values into the graphics properties before calling
    super.updateDisplayList
    backgroundColor = getStyle("backgroundColor");
    // Call super.updateDisplayList to do the rest of the work
    super.updateDisplayList(unscaledWidth, unscaledHeight);
  }
</fx:Script>
<s:Rect left="0" right="0" top="0" bottom="0">
  <s:SolidColor id="backgroundFill" />
</s:Rect>

```

Using events in custom skins

[Output: IPH, Print, Web] [Revision Control: Changing]

Most skins block user interaction to the skin. This means that you cannot trigger events defined on `InteractiveObject` on the skin. These include user interaction events such as `mouseDown`, `mouseOut`, and `click`. This is a general approach that separates the component's appearance with the component's behavior. Instead, you should use states to react to user interaction.

You can trigger events defined on `UIComponent` on the skin, however. These events include `creationComplete` and other lifecycle events.

A `Button` control's default skin defines several `Rect` elements and a `Label` control. These simple classes do not support any events. To add lifecycle event listeners, you can wrap a simple element or series of elements in a `Group` tag. This lets you register lifecycle events that are defined on `UIComponent`, but still do not allow you to register user-interaction event listeners.

The following example shows a `Group` inside the skin that triggers a `creationComplete` event:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/LifecycleEventExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" height="200" width="200">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>

  <s:Group width="150" height="100"
  >
    <s:Button label="Click It or Ticket" skinClass="mySkins.LifecycleEventSkin"/>
  </s:Group>
</s:Application>
```

The custom skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\LifecycleEventSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect id="buttonBorder" width="100%" height="20" radiusX="2" radiusY="2">
    <s:stroke>
      <mx:SolidColorStroke color="0x000000" weight="1" weight.over="2"/>
    </s:stroke>
  </s:Rect>

  <s:Group creationComplete="trace('creationComplete')">
    <s:Label id="labelDisplay"
      horizontalCenter="0" verticalCenter="1"
      left="10" right="10" top="6" bottom="2">
    </s:Label>
  </s:Group>
</s:SparkSkin>

```

Skinning Spark containers

[Output: IPH, Print, Web] [Revision Control: Changing]

Spark defines two types of containers: groups and containers. Both types of containers are in the `spark.components.*` package. Spark groups are not skinnable. Most Spark containers are skinnable. For example, the `SkinnableContainer`, `Panel`, and `TitleWindow` containers are skinnable. To be skinnable, a Spark container must be a subclass of the `SkinnableContainer` class. The `Border` container, for example, is not skinnable.

Spark groups provide a lightweight mechanism that you use to perform layout. However, Spark groups do not support skinning to ensure that they add minimal overhead to your application. To modify the visual appearance of a Spark group, you can use the corresponding Spark container, which is skinnable. By contrast, all MX containers are skinnable.

Spark container skins define a group that defines where the content children are laid out. This element has an ID of `contentGroup`. All skinnable containers have a content group. All visual children of a container are pushed into the content group and laid out using that group's layout rules.

When Flash Player applies a Spark skin to a component, it applies each graphic element in the skin class according to its order in the skin class. The first layer is added first, the next graphic element is applied second, and so on. As a result, the content group of a container skin is typically the last element in the skin class so that the content is on top of all the other layers of graphic elements.

For a list of which containers and groups are skinnable and/or scrollable, see About Spark containers.

Sizing Spark container skins

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

The most common way to skin a Spark container is to set a graphic element such as a `Rect` to the size of the container. You typically set its size by using the offset properties or the dimension properties by using one of the following methods:

- Offset (setting `top=0`, `bottom=0`, `left=0`, `right=0`)
- Dimensional (setting `height=100%`, `width=100%`)

When the components are first drawn, offset properties take precedence (so if you set them, then the dimensional properties are ignored). However, if you set the value of the offset properties, but then later explicitly set the value of the dimensional properties at runtime (for example, by setting the `height` property to 100), Flex honors the new settings.

Using offset or dimensional properties depends on the use case. In general, a skin resizes when the host component resizes. The choice of percent or constraint sizing is based on the resizing scenario. For example, if you want a label element to be always half as wide as the skin, then set the `width` property to 50%. If you want a label element to be always inset by 5 pixels, set the `left` and `right` properties to 5.

The resizable skin elements in the default Spark skins usually set the `left` and `right` properties. To specify their default size, they also set the `width` property. As a result, if the component size is not set in the application, the element's width sets the default size of the skin and the component.

If the component's size is explicitly set, the component sets the size of the element in the skin class. The `width` property is overridden by the parent size plus the values of the `left` and `right` properties.

The following example from the `ButtonSkin.mxml` class shows that the default size of the Button control is the default size of its skin. This is calculated from the rectangle in the skin, and is 69 pixels wide and 20 pixels high:

```
<s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2" radiusY="2">
  <s:stroke>
    <s:LinearGradientStroke rotation="90" weight="1">
      <s:GradientEntry color="0x000000" alpha="0.5625" alpha.down="0.6375" />
      <s:GradientEntry color="0x000000" alpha="0.75" alpha.down="0.85" />
    </s:LinearGradientStroke>
  </s:stroke>
</s:Rect>
```

Another technique for designing resizable skins is to specify a minimum size for the skin. For example, the Button skin's minimum size is 21 pixels wide by 21 pixels high. These minimums are set on the root tag of the skin class, as the following example shows:

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

Adding borders to Spark containers

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

One common use of a custom Spark skin is to add a borders to a container. Borders can be simple boxes around the perimeter of a container, or they can define drop shadows, line styles, corner radii, and other properties of a border.

To add a simple rectangular border to a container's skin, you add a Rect object. You can set the height and width of the Rect to 100% so that the skin sizes itself to the size of the container automatically.

The following example adds a simple 1-point, black line as a border around the container:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/SimpleContainerBorderExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:SkinnableContainer id="myContainer" height="200" width="200"
skinClass="mySkins.ContainerBorderSkin">
    <s:Button label="Click Me"/>
  </s:SkinnableContainer>
</s:Application>
```

The custom skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ContainerBorderSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>

  <s:Group id="contentGroup" left="0" right="0" top="0" bottom="0">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>
</s:SparkSkin>

```

The container's skin sets its offset properties (`top`, `bottom`, `left`, and `right`) to 0, so the `Rect` object is set to the same size as the container.

To create a border with rounded corners in a skin, you set the values of the `radiusX` and `radiusY` properties of the `Rect` object. These properties define the number of pixels for the corners on the x and y axes of the `Rect`. The following example sets the `radiusX` and `radiusY` properties to 10 to round the corners of the container's skin:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/RoundedContainerBorderExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:SkinnableContainer id="myContainer"
    height="200" width="200"
    skinClass="mySkins.ContainerRoundedBorderSkin">
    <s:Button label="Click Me"/>
  </s:SkinnableContainer>
</s:Application>

```

The custom skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ContainerRoundedBorderSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>

  <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
  <s:Rect left="0" right="0" top="0" bottom="0" radiusX="10" radiusY="10">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>
</s:SparkSkin>

```

You can add a border to a container by using the `Border` class. The `Border` container is a subclass of the `SkinnableContainer` class. The following example shows that the `Border` container takes the `cornerRadius`, `borderColor`, `borderVisible`, and `borderAlpha` style properties, which let you define a border similar to that shown in the previous example:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BorderContainerExample.mxml -->
<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">

  <s:Border width="200" height="200"
    borderColor="0x000000"
    borderAlpha="1"
    cornerRadius="10"
    borderWidth="1">
    <s:Button label="Click Me"/>
  </s:Border>

</s:Application>

```

Because the border-related properties are styles, you can use CSS to define a consistent border across all instances of the `Border` class; for example:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BorderContainerStyleExample.mxml -->
<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">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";

        s|Border {
            borderColor:#000000;
            borderAlpha:1;
            cornerRadius:10;
            borderWidth:1;
        }
    </fx:Style>
    <s:Border>
        <s:Button label="Click Me"/>
    </s:Border>

</s:Application>

```

To add a border to a single instance of a container or group, you are not required to create a custom skin for the container. You can instead add a Rect graphic in the application itself.

For example, you can define MXML graphic tags fragment that draws the graphic elements that make up the border. These graphic elements are typically a stroke or fill (or both).

The following example draws a border around a VGroup that contains Button controls without using a custom skin class:

```

<codeblock> Resolved code-reference.
<?xml version="1.0"?>
<!-- SparkSkinning/SimpleBorderExampleNoSkin.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Group>
        <!-- border/background graphics -->
        <s:Rect width="100%" height="100%">
            <s:stroke>
                <s:SolidColorStroke color="0x000000" weight="2"/>
            </s:stroke>
        </s:Rect>

        <!-- content of container -->
        <s:VGroup left="10" top="10" right="10" bottom="10">
            <s:Button label="Click Me"/>
            <s:Button label="Click Me"/>
        </s:VGroup>
    </s:Group>

</s:Application>

```

Adding drop shadows to Spark containers

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

To add a drop shadow to a Spark container, you use the `DropShadowFilter` class. You add the filter to one of the graphic elements defined on that container's skin class. The `DropShadowFilter` class supports the `alpha`, `angle`, `distance`, and other properties that let you customize the appearance of the drop shadow. In addition, you add a fill to the graphic element. The fill defines the color of the drop shadow.

The following example defines a drop shadow on the container's skin:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/DropShadowBorderExample.mxml -->
<mx:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:SkinnableContainer id="myContainer"
    height="200" width="200"
    skinClass="mySkins.DropShadowBorderSkin"
  >
    <s:Button label="Click Me" click="myContainer.height=100;myContainer.width=100"/>
  </s:SkinnableContainer>
</mx:Application>
```

The custom skin class for this example is as follows:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/DropShadowBorderSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>
  <!-- drop shadow -->
  <s:Rect left="0" top="0" right="0" bottom="0">
    <s:filters>
      <s:DropShadowFilter
        blurX="20" blurY="20"
        alpha="0.32"
        distance="11"
        angle="90"
        knockout="true" />
    </s:filters>
    <s:fill>
      <s:SolidColor color="0" />
    </s:fill>
```

```

</s:Rect>

<!-- layer 1: border -->
<s:Rect left="0" right="0" top="0" bottom="0">
  <s:stroke>
    <s:SolidColorStroke color="0" alpha="0.50" weight="1" />
  </s:stroke>
</s:Rect>
<!-- layer 2: background fill -->
<s:Rect id="background" left="1" top="1" right="1" bottom="1">
  <s:fill>
    <s:SolidColor color="0xFF0000" id="bgFill"/>
  </s:fill>
</s:Rect>

<s:Group id="contentGroup" left="0" right="0" top="0" bottom="0">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
</s:Group>
</s:SparkSkin>

```

You can also add a drop shadow to any individual container in the application itself. The downside to doing it this way is that the drop shadow is applied to a single instance of the container, and not all containers.

Adding padding to Spark containers

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

Another common use of Spark skins is modify the padding of Spark containers.

To add padding, you set the offset properties of the containers content group. These properties are `top`, `bottom`, `left`, and `right`.

The following example sets the offsets of the `contentGroup` to 10 so that the contents of the container are padded by 10 pixels in all directions:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/PaddedContainerBorderExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:SkinnableContainer id="myContainer"
    height="200" width="200"
    skinClass="mySkins.ContainerPaddedBorderSkin"
  >
    <s:Button label="Click Me"/>
  </s:SkinnableContainer>
</s:Application>

```

The custom skin class for this example is as follows:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ContainerPaddedBorderSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>

  <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1"/>
    </s:stroke>
  </s:Rect>
</s:SparkSkin>
```

For more information, see [Using containers](#).

Adding scroll bars to Spark containers

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

Spark containers do not have scroll bars by default. To add scroll bars to a Spark container, use the Scroller tag.

You can add a scroll bar to an instance of a Spark container by editing the container in the main application file. If you want to add scroll bars to all Spark containers, then you edit the container's skin class.

To add scroll bars to a container's skin class, you wrap the content group or other element in a Scroller tag. The child of a Scroller tag must implement the `IViewport` interface. This interface is implemented by `Group` and `DataGroup`, and some components such as the `RichEditableText` control.

To ensure that scroll bars appear, do not set explicit sizes on the group. If you set the size of the group explicitly, then that size becomes the size of the `Group` and no scroll bars will be shown. Instead, use constraints to size the Scroller (such as setting the `height` property to 100%). If the scroller is unconstrained, it sizes itself to be as big as the viewport. When you constrain the size of the Scroller to something that cannot fit all the `Group` content, the container displays scrollers so that you can view all of the contents.

The following example defines the size of the application, and sizes the container to a percent of the overall height and width. The scroller is constrained to the size of the viewport (in this case, the content group). Because the image is larger than the size of its parent, Flex displays scroll bars.

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ScrollbarContainerExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" height="200" width="200">
  <s:SkinnableContainer id="myContainer"
    height="50%" width="50%"
    skinClass="mySkins.ScrollBarContainerSkin"
  >
    <mx:Image source="@Embed(source='../assets/myImage.jpg')"/>
  </s:SkinnableContainer>
</s:Application>
```

This example uses the following custom skin class:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ScrollBarContainerSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Scroller>
    <s:Group id="contentGroup">
      <s:layout>
        <s:BasicLayout/>
      </s:layout>
    </s:Group>
  </s:Scroller>
</s:SparkSkin>
```

The Scroller layout does not support the left/right/top/bottom constraints on the content group of the container. And in general, you should not explicitly set the `height` and `width` properties of the Scroller's content group.

You can use the `minViewportInset` property on the Scroller class to inset the viewport relative to its Scroller along the edges where a scroll bar does not already keep the viewport away from the edge. For example, if you set the `minViewportInset` property to 10, then the right edge of the viewport would be 10 pixels from the right edge of its scroller, unless the vertical scroll bar was visible. When a scroll bar is visible, the viewport and the scroll bar would not have any space between them.

For more information, see [Scrolling Spark containers](#).

Adding background fills and images to Spark containers

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

To add background fills and background images to a Spark container's skin, you add a `Rect` or other graphic element and add the fill to that element.

To add a color fill, you add a graphic element that is a subclass of `FilledElement` and define a child fill tag. Subclasses of `FilledElement` include `Rect`, `Ellipse`, and `Path`. The fill must be an `IFill`, which can be a `BitmapFill`, `LinearGradient`, `RadialGradient`, or `SolidColor`.

The following example defines three containers with three different fills:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BackgroundFillExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <s:SkinnableContainer id="myContainer"
    height="200" width="200"
    skinClass="mySkins.BackgroundFillSkin"
  >
    <s:Button label="Basic Fill"/>
  </s:SkinnableContainer>
  <s:SkinnableContainer id="myContainer2"
    height="200" width="200"
    skinClass="mySkins.RadialBackgroundFillSkin"
  >
    <s:Button label="Radial Gradient Fill"/>
  </s:SkinnableContainer>
  <s:SkinnableContainer id="myContainer3"
    height="200" width="200"
    skinClass="mySkins.LinearBackgroundFillSkin"
  >
    <s:Button label="Linear Gradient Fill"/>
  </s:SkinnableContainer>
</s:Application>
```

The example uses the following custom skin class for the radial gradient fill:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/RadialBackgroundFillSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>
  <!-- layer 1: border -->
  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:stroke>
      <s:SolidColorStroke color="0" alpha="0.50" weight="1" />
    </s:stroke>
  </s:Rect>
  <!-- background fill -->
  <s:Rect id="background" left="1" top="1" right="1" bottom="1">
    <s:fill>
      <s:RadialGradient>
        <s:entries>
          <s:GradientEntry color="0xFFAABB" ratio="0" alpha="1"/>
          <s:GradientEntry color="0xFFCCDD" ratio=".33" alpha="1"/>
          <s:GradientEntry color="0xFFEEFF" ratio=".66" alpha="1"/>
        </s:entries>
      </s:RadialGradient>
    </s:fill>
  </s:Rect>

  <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
</s:SparkSkin>

```

The example uses the following custom skin class for the basic background fill:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/BackgroundFillSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>
  <!-- layer 1: border -->
  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:stroke>
      <s:SolidColorStroke color="0" alpha="0.50" weight="1" />
    </s:stroke>
  </s:Rect>
  <!-- background fill -->
  <s:Rect id="background" left="1" top="1" right="1" bottom="1">
    <s:fill>
      <s:SolidColor id="bgFill" color="0xFF0000"/>
    </s:fill>
  </s:Rect>

  <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
</s:SparkSkin>
```

The example uses the following custom skin class for the linear background fill:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/LinearBackgroundFillSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>
  <!-- layer 1: border -->
  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:stroke>
      <mx:SolidColorStroke color="0" alpha="0.50" weight="1"/>
    </s:stroke>
  </s:Rect>
  <!-- background fill -->
  <s:Rect id="background" left="1" top="1" right="1" bottom="1">
    <s:fill>
      <s:LinearGradient>
        <s:entries>
          <s:GradientEntry color="0xFFAABB" ratio="0" alpha="1"/>
          <s:GradientEntry color="0xFFCCDD" ratio=".33" alpha="1"/>
          <s:GradientEntry color="0xFFFFFFFF" ratio=".66" alpha="1"/>
        </s:entries>
      </s:LinearGradient>
    </s:fill>
  </s:Rect>

  <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
</s:SparkSkin>

```

You can also add a background fill to a content group, although the content group often defines an offset or padding properties. These constraints are applied to the background in addition to the content. As a result, you typically add the background fill to a graphic element that is set to the entire height and width of the container.

To add a background image, you use the Bitmap fill, as the following example shows:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BackgroundFillExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <s:SkinnableContainer id="myContainer"
    height="256" width="206"
    skinClass="mySkins.BitmapFillBackgroundSkin"
  >
    <s:Button label="Bitmap Fill"/>
  </s:SkinnableContainer>
</s:Application>
```

This example uses the following custom skin class:

```
<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/BitmapFillBackgroundSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">
  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>
  <!-- layer 1: border -->
  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:stroke>
      <s:SolidColorStroke color="0" alpha="0.50" weight="3" />
    </s:stroke>
  </s:Rect>
  <!-- background fill -->
  <s:Rect id="background" left="3" top="3" right="3" bottom="3" alpha=".25">
    <s:fill>
      <s:BitmapFill source="@Embed(source='../assets/myImage.jpg')"/>
    </s:fill>
  </s:Rect>

  <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
  </s:Group>
</s:SparkSkin>
```

In this example, the container is perfectly sized to fit the bitmap background plus the width of the border. In many cases, though, you will have to set properties on the fill to avoid tiling or apply clipping. For more information, see the `BitmapFill` class in the *ActionScript Language Reference*.

The content group of the container's skin is typically the last element defined in the skin class. This is because the skin's layers are applied in the order in which they appear in the skin class. When a skin class has a background fill in a skin class, you add the content layer after it so that the content is not covered by the fill.

Unloading skins

[Output: IPH, Print, Web] [Revision Control: Changing]

You cannot unload or remove a skin class from its host component without specifying another skin to take its place. Otherwise, the component would have no definition of its visual parts (and would therefore break the skinning contract that requires the skin class to declare all skin states).

To unload a custom skin, you typically load the default skin class in its place. To do this, you set the component's `skinClass` property to another skin class.

The following example applies a custom skin to the button when the application starts up. It then toggles the custom skin with the Button control's default skin class when you click the button:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/SimpleUnloadExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        import spark.skins.spark.ButtonSkin;
        import mySkins.MyButtonSkin;
        import flash.utils.*;

        private function toggleSkin():void {
            if (getQualifiedClassName(myButton.getStyle("skinClass")) ==
getQualifiedClassName(MyButtonSkin)) {
                myButton.setStyle("skinClass", Class(ButtonSkin));
            } else {
                myButton.setStyle("skinClass", Class(ButtonSkin));
            }
        }
    </fx:Script>
    <s:Button id="myButton"
        skinClass="mySkins.MyButtonSkin"
        label="Toggle Skin"
        color="0xFFFFFF"
        click="toggleSkin()" />
</s:Application>
```

When you unload a skin, Flex calls the component's `detachSkin()` method. You typically do not call this method explicitly. This method then calls the `partRemoved()` method on all skin parts (static, dynamic, or deferred). You typically do not call this method explicitly either.

If the skin has been removed from the display list, then the `detachSkin()` method will not be called. This method is only called when the skin class is explicitly changed or removed.

Transitions with Spark skins

[Output: IPH, Print, Web] [Revision Control: Changing]

You can use transitions to add visual appeal to your Spark skins. Transitions are triggered off of state changes, which are explicitly supported in Spark skins. You should avoid using effects in skin classes, on the other hand, because they are typically triggered off of user interaction, which are not commonly supported in Spark skins.

You can use transitions in Spark skins in the same way you would use them in your application. You add a `<s:transitions>` tag as a child tag of the skin's root tag. You then define the transitions and which states they apply to with the `toState` and `fromState` on the `<s:Transition>` child tags. Because the skin is notified of state changes from the host component, you do not have to add any logic to support state changes to the skin.

The following example uses the `Resize` transition to grow and shrink the `Button` control's label and border. The transitions are triggered when the user rolls the pointer over the `Button` control and when the user rolls the pointer off of the `Button` control.

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ButtonTransitionExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="200" width="200">

    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:SkinnableContainer id="myContainer"
        height="200" width="200">
        <s:Button
            label="Click Me"
            skinClass="mySkins.ButtonTransitionSkin"/>
    </s:SkinnableContainer>

</s:Application>
```

The custom skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\ButtonTransitionSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21">
  <s:transitions>
    <s:Transition fromState="up" toState="over">
      <s:Sequence target="{labelDisplay, buttonBorder}">
        <s:Resize
          widthBy="10"
        />
      </s:Sequence>
    </s:Transition>
    <s:Transition fromState="over" toState="up">
      <s:Sequence target="{labelDisplay, buttonBorder}">
        <s:Resize
          widthBy="-10"
        />
      </s:Sequence>
    </s:Transition>
  </s:transitions>
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <s:Rect id="buttonBorder" width="69" height="20" radiusX="2" radiusY="2">
    <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="1" weight.over="2"/>
    </s:stroke>
  </s:Rect>

  <!-- layer 8: text -->
  <s:Label id="labelDisplay"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="6" bottom="2">
  </s:Label>
</s:SparkSkin>

```

When using transitions inside Spark skins, you might have to assign IDs to elements that do not by default have IDs. This is because transitions require a target when you define them, and without an ID, you cannot target a specific element.

As with standard Flex controls, you can use transitions with multiple effects that are playing in parallel on a Spark skin. The following example resizes and changes the color of the button's label when the user moves their mouse over or out of the Button:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ButtonParallelTransitionExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  height="200" width="200">

  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <s:Group width="150" height="100">
    <s:Button label="Click It or Ticket" skinClass="mySkins.ButtonParallelTransitionSkin"/>
  </s:Group>

</s:Application>

```

The custom skin class for this example is as follows:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ButtonParallelTransitionSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21"
  alpha.disabled="0.5"
  creationComplete="initSkin()">
  <s:transitions>
    <s:Transition fromState="up" toState="over">
      <s:Parallel>
        <s:Animate target="{labelDisplay}">
          <s:SimpleMotionPath
            property="fontSize"
            valueFrom="{originalFontSize}"
            valueTo="{largerFontSize}"/>
        </s:Animate>
        <s:AnimateColor id="colorUp" target="{labelDisplay}">
          <s:SimpleMotionPath
            property="color"
            valueFrom="0x000000"
            valueTo="0xFF0000"/>
        </s:AnimateColor>
      </s:Parallel>
    </s:Transition>
    <s:Transition fromState="over" toState="up">
      <s:Parallel>
        <s:Animate target="{labelDisplay}">
          <s:SimpleMotionPath
            property="fontSize"
            valueFrom="{largerFontSize}"
            valueTo="{originalFontSize}"/>
        </s:Animate>
        <s:AnimateColor id="colorDown" target="{labelDisplay}" duration="1500">
          <s:SimpleMotionPath

```

```

        property="color"
        valueFrom="0xFF0000"
        valueTo="0x000000"/>
    </s:AnimateColor>
</s:Parallel>
</s:Transition>
</s:transitions>
<fx:Metadata>
    [HostComponent("spark.components.Button")]
</fx:Metadata>

<fx:Script>
    static private const exclusions:Array = ["labelDisplay"];
    override public function get colorizeExclusions():Array {return exclusions;}

    [Bindable]
    public var originalFontSize:Number;
    [Bindable]
    public var largerFontSize:Number;

    private function initSkin():void {
        originalFontSize = hostComponent.getStyle("fontSize");
        largerFontSize = originalFontSize + 4;
    }
</fx:Script>

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

<s:Rect left="-1" right="-1" top="-1" bottom="-1" radiusX="2" radiusY="2">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry color="0x000000"
                color.down="0xFFFFFFFF"
                alpha="0.01"
                alpha.down="0" />
            <s:GradientEntry color="0x000000"
                color.down="0xFFFFFFFF"
                alpha="0.07"
                alpha.down="0.5" />
        </s:LinearGradient>
    </s:fill>
</s:Rect>

<s:Rect left="1" right="1" top="1" bottom="1" radiusX="2" radiusY="2">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry color="0xFFFFFFFF"
                color.over="0xBBBDBD"
                color.down="0xAAAAAA"
                alpha="0.85" />
            <s:GradientEntry color="0xD8D8D8"
                color.over="0x9FA0A1"

```

```

        color.down="0x929496"
        alpha="0.85" />
    </s:LinearGradient>
</s:fill>
</s:Rect>

<s:Rect left="1" right="1" bottom="1" height="9" radiusX="2" radiusY="2">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry color="0x000000" alpha="0.0099" />
            <s:GradientEntry color="0x000000" alpha="0.0627" />
        </s:LinearGradient>
    </s:fill>
</s:Rect>

<s:Rect left="1" right="1" top="1" height="9" radiusX="2" radiusY="2">
    <s:fill>
        <s:SolidColor color="0xFFFFFFFF"
            alpha="0.33"
            alpha.over="0.22"
            alpha.down="0.12" />
    </s:fill>
</s:Rect>

<s:Rect left="1" right="1" top="1" bottom="1"
    radiusX="2" radiusY="2" excludeFrom="down">
    <s:stroke>
        <s:LinearGradientStroke rotation="90" weight="1">
            <s:GradientEntry color="0xFFFFFFFF" alpha.over="0.22" />
            <s:GradientEntry color="0xD8D8D8" alpha.over="0.22" />
        </s:LinearGradientStroke>
    </s:stroke>
</s:Rect>

<s:Rect left="1" top="1" bottom="1" width="1" includeIn="down">
    <s:fill>
        <s:SolidColor color="0x000000" alpha="0.07" />
    </s:fill>
</s:Rect>

<s:Rect right="1" top="1" bottom="1" width="1" includeIn="down">
    <s:fill>
        <s:SolidColor color="0x000000" alpha="0.07" />
    </s:fill>
</s:Rect>

<s:Rect left="2" top="1" right="2" height="1" includeIn="down">
    <s:fill>
        <s:SolidColor color="0x000000" alpha="0.25" />
    </s:fill>
</s:Rect>

<s:Rect left="1" top="2" right="1" height="1" includeIn="down">
    <s:fill>
        <s:SolidColor color="0x000000" alpha="0.09" />
    </s:fill>
</s:Rect>

<s:Rect left="0" right="0" top="0" bottom="0"
    width="69" height="20"

```

```

        radiusX="2" radiusY="2">
        <s:stroke>
            <s:LinearGradientStroke rotation="90" weight="1">
                <s:GradientEntry color="0x000000"
                    alpha="0.5625"
                    alpha.down="0.6375" />
                <s:GradientEntry color="0x000000"
                    alpha="0.75"
                    alpha.down="0.85" />
            </s:LinearGradientStroke>
        </s:stroke>
    </s:Rect>
    <s:Label id="labelDisplay"
        textAlign="center"
        verticalAlign="middle"
        lineBreak="toFit"
        maxDisplayedLines="1"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:SparkSkin>

```

You can use transitions and states to effectively add and remove elements from the skin. For example, if you want an image to appear on a Button control during a mouse over, you can change the values of properties such as `alpha` and `height` and `width` from 0 to a new value. When the user mouses out, you can change the values of the `alpha` and `height` and `width` properties back to 0.

The following example displays an image of a butterfly when you move the mouse over the Button control, and removes the image when you move the mouse away:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ButterflySkinExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    backgroundColor="0x999999">

    <s:Button id="myButton"
        fontWeight="bold"
        color="0xFFFFFFFF"
        label="Bug of the Day"
        skinClass="mySkins.ButterflySkin"/>

</s:Application>

```

The custom skin class for this example is as follows:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- The default skin class for the Spark Button component.

    @langversion 3.0
    @playerversion Flash 10
    @playerversion AIR 1.5
    @productversion Flex 4
-->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21" alpha.disabled="0.5">

    <!-- host component -->
    <fx:Metadata>
        <![CDATA[
            /**
             * @copy spark.skins.spark.ApplicationSkin#hostComponent
             */
            [HostComponent("spark.components.Button")]
        ]]>
    </fx:Metadata>

    <s:transitions>
        <mx:Transition fromState="up" toState="over">
            <mx:Parallel>
                <s:Resize target="{ myImage }" heightFrom="0" heightTo="90" duration="200"/>
                <s:Fade target="{ myImage }" alphaFrom="0" alphaTo="1" duration="200"/>
                <s:Fade target="{ labelDisplay }" alphaFrom="1" alphaTo="0" duration="200"/>
            </mx:Parallel>
        </mx:Transition>
        <mx:Transition fromState="over" toState="up">
            <mx:Parallel>
                <s:Fade target="{ myImage }" alphaFrom="1" alphaTo="0" duration="150"/>
                <s:Resize target="{ myImage }" heightFrom="90" heightTo="0" duration="150"/>
                <s:Fade target="{ labelDisplay }" alphaFrom="0" alphaTo="1" duration="150"/>
            </mx:Parallel>
        </mx:Transition>
    </s:transitions>

    <fx:Script>
        <![CDATA[
            /* Define the skin elements that should not be colorized.
             For button, the graphics are colorized but the label is not. */
            static private const exclusions:Array = ["labelDisplay"];

            /**
             * @private
             */
            override public function get colorizeExclusions():Array {return exclusions;}

            /**
             * @private
             */
            override protected function initializationComplete():void

```

```

    {
        useBaseColor = true;
        super.initializationComplete();
    }

    /**
     * @private
     */
    override protected function updateDisplayList(unscaledWidth:Number,
    unscaledHeight:Number) : void
    {
        var cr:Number = getStyle("cornerRadius");
        super.updateDisplayList(unscaledWidth, unscaledHeight);
    }

    private var cornerRadius:Number = 2;
    ]]>
</fx:Script>

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

    <s:Rect id="blueRect" radiusX="8" radiusY="8" top="0" right="0" bottom="0" left="0"
    minHeight="30">
        <s:fill>
            <s:LinearGradient x="0" y="0" scaleX="44" rotation="90">
                <s:GradientEntry color="#3399ff" ratio="0" color.over="#66CCFF"/>
                <s:GradientEntry color="#3366cc" ratio="1" color.over="#3399CC"/>
            </s:LinearGradient>
        </s:fill>
        <s:stroke>
            <s:SolidColorStroke color="ffffff" weight="2"/>
        </s:stroke>
    </s:Rect>

    <!-- Border -->
    <s:Rect
        radiusX="6"
        radiusY="6"
        top="2"
        right="2"
        height="15"
        left="2">
        <s:fill>
            <s:LinearGradient x="0" y="0" scaleX="23" rotation="90">
                <s:GradientEntry color="ffffff" ratio="0" alpha=".3"/>
                <s:GradientEntry color="ffffff" ratio="1" alpha=".1"/>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

```

```
        </s:LinearGradient>
    </s:fill>
</s:Rect>

    <mx:Image
        id="myImage"
        alpha="0"
        height="0"
        source="@Embed(source='../assets/butterfly.png')"
        left="20"/>

    <s:Label id="labelDisplay"
        textAlign="center"
        verticalAlign="middle"
        maxDisplayedLines="1"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2">
</s:Label>

</s:SparkSkin>
```

For more information, see [Using Transitions](#) and [Using Spark effects](#).

Using FXG with Spark skins

[Output: IPH, Print, Web] [Revision Control: Changing]

Spark skins can use documents created by external tools that support the FXG file format. FXG is an XML-based language for interchanging graphics among different applications that support it. For skinning with FXG, you typically generate FXG from an external tool and either use the output as a custom component, or copy the elements into an MXML skin class file. There is very little conversion necessary to convert FXG syntax to MXML graphics tags.

An FXG file consists of a single definition and an optional library that is contained within a root `<Graphic>` element. The elements in an FXG file are defined by the FXG language specification. An FXG document fragment can include zero or more containers and graphic elements. An FXG file can be a standalone `*.fxg`. You can also mix FXG syntax inline in your MXML files. For more information, see [Flash XML Graphics \(FXG\)](#).

You can use FXG in your skins in one of two ways:

- Inline (copy and paste the FXG elements into an MXML file)
- Custom component (reference an `*.fxg` file from your MXML)

If you use an `*.fxg` file as a component, the FXG is optimized for memory. The down side is that the FXG is not interactive. You cannot do things like invoke events or add ActionScript to it. You cannot set properties or execute methods on the objects within the FXG file.

If you copy and paste FXG output into an MXML file (such as a skin), the output is not optimized. Instead, the Flex compiler converts the FXG syntax to the equivalent MXML graphics tags. For example, a `<Path>` element in an FXG file becomes an instance of a `mx.graphics.Path` class in your skin class. The result is that you can then add interactivity to the MXML graphics, but you do not get the benefit of memory optimization.

The FXG language namespace is "http://ns.adobe.com/fxg/2008". Most of the elements in this namespace have ActionScript backing implementations. What this means is that most tags in the FXG language correspond to ActionScript classes with the same name. For example, if your FXG file uses a <LinearGradient> element, there is an equivalent class named LinearGradient in the Flex SDK.

There is one exception, TextGraphic. This tag in FXG roughly corresponds to the RichText class in the Flex SDK, but has slightly different functionality.

Using FXG syntax inline in Spark skins

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

You can use any number of FXG elements inline in a Spark skin.

To use FXG elements inline, you copy and paste the <Graphic> element and its contents from the FXG file into your Spark skin. These elements are then interpreted by the Flex compiler as MXML graphics tags.

Use the following general guidelines when working with FXG files inline:

- Remove the namespace declarations from the <Graphic> element and integrate them with your skin's namespace declarations in the <s:Skin> or <s:SparkSkin> tag.

If the FXG fragment uses namespaces other than those in the Spark skin, you must add those namespace declarations to the skin's root element. For example, an FXG file exported from Adobe Illustrator includes the following namespaces:

```
m1ns:ai="http://ns.adobe.com/ai/2008"  
xmlns:d="http://ns.adobe.com/fxg/2008/dt"
```

- Use the Spark namespace identifier for all FXG elements. For example, replace the <Graphic> element with <s:Graphic>.
- If the FXG file uses symbols or assets defined in its <Library> element, you must add those definitions to your skin's <fx:Library> tag. This must be a child of the Spark skin's root tag.
- Do not include the FXG language namespace (http://ns.adobe.com/fxg/2008) in your Spark skin. The elements in this namespace are included in the default Flex namespace and there can be only one language namespace per document.
- You do not have to remove the <Private> element from the FXG fragment (although you should convert the <Private> element to the <s:Private> tag by adding the namespace identifier. The Flex compilers ignore the contents of this element.
- Replace the <TextGraphic> element with the <s:RichText> tag.

The following file, exported in FXG format from Illustrator, includes the private information as well as the <Graphic> root element. The contents of this *.fxg file are being shown so that you can then see in subsequent example files which parts should be copied and pasted into the skin.

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8" ?>
<!-- assets/skins/SimpleBox.fyg -->
<Graphic version="1.0" viewHeight="30" viewWidth="100" ai:appVersion="14.0.0.367" d:id="1"
xmlns="http://ns.adobe.com/fyg/2008" xmlns:ai="http://ns.adobe.com/ai/2008"
xmlns:d="http://ns.adobe.com/fyg/2008/dt">
  <Library/>
  <Group x="-0.296875" y="-0.5" d:id="2" d:type="layer" d:userLabel="Layer 1">
    <Group d:id="3">
      <Rect x="0.5" y="0.5" width="100" height="30" ai:knockout="0">
        <fill>
          <LinearGradient x="0.5" y="15.5" scaleX="100" rotation="-0">
            <GradientEntry color="#ffffff" ratio="0"/>
            <GradientEntry ratio="1"/>
          </LinearGradient>
        </fill>
        <stroke>
          <SolidColorStroke color="#0000ff" caps="none" weight="1" joints="miter"
miterLimit="4"/>
        </stroke>
      </Rect>
    </Group>
  </Group>
  <Private>
    <ai:PrivateElement d:ref="#1">
      <ai:SaveOptions>
        <ai:Dictionary>
          <ai:DictEntry name="preserveGradientPolicy" value="3" valueType="Integer"/>
          <ai:DictEntry name="rasterizeResolution" value="72" valueType="Integer"/>
          <ai:DictEntry name="clipToActiveArtboard" value="1" valueType="Boolean"/>
          <ai:DictEntry name="downsampleLinkedImages" value="0" valueType="Boolean"/>
          <ai:DictEntry name="preserveFilterPolicy" value="3" valueType="Integer"/>
          <ai:DictEntry name="preserveTextPolicy" value="3" valueType="Integer"/>
          <ai:DictEntry name="writeImages" value="1" valueType="Boolean"/>
          <ai:DictEntry name="includeXMP" value="0" valueType="Boolean"/>
          <ai:DictEntry name="aiEditCap" value="1" valueType="Boolean"/>
          <ai:DictEntry name="versionKey" value="1" valueType="Integer"/>
          <ai:DictEntry name="includeSymbol" value="0" valueType="Boolean"/>
        </ai:Dictionary>
      </ai:SaveOptions>
      <ai:DocData base="SimpleBox.assets/images"/>
      <ai:Artboards originOffsetH="0" originOffsetV="30" rulerCanvasDiffH="50.5"
rulerCanvasDiffV="-14.5" zoom="17.17">
        <ai:Artboard active="1" index="0" right="100" top="30"/>
        <ai:ArtboardsParam all="0" range="" type="0"/>
      </ai:Artboards>
    </ai:PrivateElement>
    <ai:PrivateElement d:ref="#2">
      <ai:LayerOptions colorType="ThreeColor">
        <ai:ThreeColor blue="257" green="128.502" red="79.31"/>
      </ai:LayerOptions>
    </ai:PrivateElement>
    <ai:PrivateElement ai:hashCode="769d7bac08ad6bdcf80f40fca11df6c0" d:ref="#3">
      <ai:Rect height="30" knockout="0" width="100" x="0.5" y="0.5">

```

```

    <ai:Stroke colorType="ThreeColor" miterLimit="4" weight="1">
      <ai:ThreeColor blue="1"/>
    </ai:Stroke>
    <ai:Fill colorType="Gradient">
      <ai:Gradient gradientType="linear" length="100" originX="0.5" originY="15.5">
        <ai:GradientStops>
          <ai:GradientStop colorType="GrayColor" rampPoint="0">
            <ai:GrayColor/>
          </ai:GradientStop>
          <ai:GradientStop colorType="GrayColor" rampPoint="100">
            <ai:GrayColor gray="1"/>
          </ai:GradientStop>
        </ai:GradientStops>
      </ai:Gradient>
    </ai:Fill>
    <ai:ArtStyle/>
  </ai:Rect>
</ai:PrivateElement>
</Private>
</Graphic>

```

The following example uses a component that has a Spark skin based on the FXG output file:

```

<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/FXGButtonExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" height="200" width="200">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <s:SkinnableContainer id="myContainer"
    height="200" width="200"
  >
    <s:Button label="Click Me" skinClass="mySkins.FXGButtonSkin"/>
  </s:SkinnableContainer>
</s:Application>

```

The following code is the custom skin used in this example. This example simplifies the FXG fragment by removing the <Private> element, and integrating the namespaces.

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\FXGButtonSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:ai="http://ns.adobe.com/ai/2008"
  xmlns:d="http://ns.adobe.com/fxg/2008/dt"
  minWidth="21" minHeight="21">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <!-- FXG exported from Adobe Illustrator. -->
  <s:Graphic version="1.0" viewHeight="30" viewWidth="100" ai:appVersion="14.0.0.367"
d:id="1">
    <s:Group x="-0.296875" y="-0.5" d:id="2" d:type="layer" d:userLabel="Layer 1">
      <s:Group d:id="3">
        <s:Rect x="0.5" y="0.5" width="100" height="30" ai:knockout="0">
          <s:fill>
            <s:LinearGradient x="0.5" y="15.5" scaleX="100" rotation="-0">
              <s:GradientEntry color="#ffffff" ratio="0"/>
              <s:GradientEntry ratio="1"/>
            </s:LinearGradient>
          </s:fill>
          <s:stroke>
            <s:SolidColorStroke color="#0000ff" caps="none" weight="1" joints="miter"
miterLimit="4"/>
          </s:stroke>
        </s:Rect>
      </s:Group>
    </s:Group>
  </s:Graphic>
  <s:Label id="labelDisplay"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="15" bottom="2">
  </s:Label>
</s:SparkSkin>

```

You can optionally remove the <Private> tag and its contents from the FXG fragment. Some tools export FXG with additional syntax so that the file can be re-edited. An FXG document exported from Illustrator, for example, includes a <Private> block of code that is used by Adobe® Illustrator® if you want to edit the file again. If you don't want to edit the file in Illustrator again, then you can remove the <Private> block.

Using FXG files as custom components in Spark skins

[Chunk: No] [Output: IPH, Print, Web] [Revision Control: Changing]

You can use an FXG file as a custom component to define the look of your custom Spark skin. You must still define skin parts, states, and other interactivity in your MXML skin class, but the FXG can define graphic elements that are used by that skin.

In most cases, you do not have to edit the FXG document before using it with a Spark skin. The FXG file includes a valid XML identifier, and includes all namespace definitions that it needs in its root tag.

The FXG file can must have a *.fxg filename suffix. You cannot have a file of the same name with the *.mxml suffix in the same directory.

When using an FXG component in a Spark skin class, you can leave the <Private> element in the file. The Flex compilers check the Private block for validity, but do not attempt to render any information in that block.

The location of a custom component used in a skin file is relative to the application, module, or component that uses the skin, not to the skin file. For example, if you have the following structure:

```
/app-dir/MainApp.mxml  
/app-dir/skins/CustomSkin.mxml
```

The FXG custom component used in the skin would be in a directory that is parallel to the skins directory. For example:

```
app-dir/comps/FXGComponent.fyg
```

In the skin file, you declare the comps namespace as comps.*.

The following example uses a custom FXG component in its custom skin:

```
<codeblock> Resolved code-reference.  
<?xml version="1.0" encoding="utf-8"?>  
<!-- SparkSkinning/FXGCompButtonExample.mxml -->  
<s:Application  
  xmlns:fx="http://ns.adobe.com/mxml/2009"  
  xmlns:mx="library://ns.adobe.com/flex/halo"  
  xmlns:s="library://ns.adobe.com/flex/spark" height="200" width="200">  
  <s:layout>  
    <s:HorizontalLayout/>  
  </s:layout>  
  <s:SkinnableContainer id="myContainer"  
    height="200" width="200"  
  >  
    <s:Button label="Click Me" skinClass="mySkins.FXGCompButtonSkin"/>  
  </s:SkinnableContainer>  
</s:Application>
```

The following custom skin uses the BoxComp.fyg file as a custom component that draws the box in the skin:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\FXGCompButtonSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="21" minHeight="21"
  xmlns:comps="comps.*">
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>
  <s:states>
    <s:State name="up"/>
    <s:State name="over"/>
    <s:State name="down"/>
    <s:State name="disabled"/>
  </s:states>
  <comps:BoxComp/>
  <s:Label id="labelDisplay"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="15" bottom="2">
  </s:Label>
</s:SparkSkin>

```

The following FXG file was directly exported from Illustrator. It defines a bordered box and gradient fill:

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8" ?>
<!-- SparkSkinning/comps/FXGCompButtonExample.mxml -->
<fxg:Graphic version="1.0"
  viewHeight="30" viewWidth="100"
  ai:appVersion="14.0.0.367"
  d:id="1"
  xmlns:fxg="http://ns.adobe.com/fxg/2008"
  xmlns:ai="http://ns.adobe.com/ai/2008"
  xmlns:d="http://ns.adobe.com/fxg/2008/dt">
  <fxg:Library/>
  <fxg:Group x="-0.296875" y="-0.5" d:id="2" d:type="layer" d:userLabel="Layer 1">
    <fxg:Group d:id="3">
      <fxg:Rect x="0.5" y="0.5" width="100" height="30" ai:knockout="0">
        <fxg:fill>
          <fxg:LinearGradient x="0.5" y="15.5" scaleX="100" rotation="-0">
            <fxg:GradientEntry color="#ffffff" ratio="0"/>
            <fxg:GradientEntry ratio="1"/>
          </fxg:LinearGradient>
        </fxg:fill>
        <fxg:stroke>
          <fxg:SolidColorStroke color="#0000ff" caps="none" weight="1" joints="miter"
            miterLimit="4"/>
        </fxg:stroke>
      </fxg:Rect>
    </fxg:Group>
  </fxg:Group>
  <fxg:Private>

```

```

<ai:PrivateElement d:ref="#1">
  <ai:SaveOptions>
    <ai:Dictionary>
      <ai:DictEntry name="preserveGradientPolicy" value="3" valueType="Integer"/>
      <ai:DictEntry name="rasterizeResolution" value="72" valueType="Integer"/>
      <ai:DictEntry name="clipToActiveArtboard" value="1" valueType="Boolean"/>
      <ai:DictEntry name="downsampleLinkedImages" value="0" valueType="Boolean"/>
      <ai:DictEntry name="preserveFilterPolicy" value="3" valueType="Integer"/>
      <ai:DictEntry name="preserveTextPolicy" value="3" valueType="Integer"/>
      <ai:DictEntry name="writeImages" value="1" valueType="Boolean"/>
      <ai:DictEntry name="includeXMP" value="0" valueType="Boolean"/>
      <ai:DictEntry name="aiEditCap" value="1" valueType="Boolean"/>
      <ai:DictEntry name="versionKey" value="1" valueType="Integer"/>
      <ai:DictEntry name="includeSymbol" value="0" valueType="Boolean"/>
    </ai:Dictionary>
  </ai:SaveOptions>
  <ai:DocData base="SimpleBox.assets/images"/>
  <ai:Artboards originOffsetH="0" originOffsetV="30" rulerCanvasDiffH="50.5"
rulerCanvasDiffV="-14.5" zoom="17.17">
    <ai:Artboard active="1" index="0" right="100" top="30"/>
    <ai:ArtboardsParam all="0" range="" type="0"/>
  </ai:Artboards>
</ai:PrivateElement>
<ai:PrivateElement d:ref="#2">
  <ai:LayerOptions colorType="ThreeColor">
    <ai:ThreeColor blue="257" green="128.502" red="79.31"/>
  </ai:LayerOptions>
</ai:PrivateElement>
<ai:PrivateElement ai:hashCode="769d7bac08ad6bdcf80f40fca11df6c0" d:ref="#3">
  <ai:Rect height="30" knockout="0" width="100" x="0.5" y="0.5">
    <ai:Stroke colorType="ThreeColor" miterLimit="4" weight="1">
      <ai:ThreeColor blue="1"/>
    </ai:Stroke>
    <ai:Fill colorType="Gradient">
      <ai:Gradient gradientType="linear" length="100" originX="0.5" originY="15.5">
        <ai:GradientStops>
          <ai:GradientStop colorType="GrayColor" rampPoint="0">
            <ai:GrayColor/>
          </ai:GradientStop>
          <ai:GradientStop colorType="GrayColor" rampPoint="100">
            <ai:GrayColor gray="1"/>
          </ai:GradientStop>
        </ai:GradientStops>
      </ai:Gradient>
    </ai:Fill>
  </ai:Rect>
</ai:PrivateElement>
</fxg:Private>
</fxg:Graphic>

```

For more information, see Using FXG.

Subcomponent skinning

[Output: IPH, Print, Web] [Revision Control: Changing]

Some components are composites of other components. They use other components as part of their user interface. For example, a `NumericStepper` consists of a `Button` control with a down arrow, a `Button` control with an up arrow, and a `TextInput` control that displays the current value. The components that make up a composite component's user interface are known as subcomponents.

To skin Spark subcomponents you edit the skin parts in the skin class. Each subcomponent corresponds to a skin part that is defined on the host component. For example, to customize the button skin parts in a `NumericStepperSkin` class, you can set the value of their `skinClass` property to a custom class.

In many cases, the subcomponents that are defined in the main skin have their own skin classes. This is common if the subcomponents need to define their own appearance based on their state. For example, the buttons in a `NumericStepper` should react to their own `up` and `down` states rather than the `up` and `down` states of the composite component.

The following example defines custom subcomponent skins for the up and down buttons in a `NumericStepper` composite skin:

```
<codeblock> Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/NumericStepperSkinPartExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="200" width="200">

    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:SkinnableContainer id="myContainer"
        height="200" width="200">
        <s:NumericStepper skinClass="mySkins.NumericStepperSkin"/>
    </s:SkinnableContainer>
</s:Application>
```

The following composite skin class is nearly identical to the default `NumericStepperSkin` except that it defines custom skins for the button subcomponents. In this case, the `spark.skins.spark.SpinnerDecrementButtonSkin` and `spark.skins.spark.SpinnerIncrementButtonSkin` classes are replaced with `mySkins.SpinnerDownButton` and `mySkins.SpinnerUpButton` respectively.

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/NumericStepperSkin.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minHeight="24"
  alpha.disabled="0.5">

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

  <fx:Script>
    /* Define the skin elements that should not be colorized.
    For numeric stepper, the skin itself is colorized but the individual parts are not. */
    static private const exclusions:Array = ["textInput", "decrementButton",
"incrementButton"];
    override public function get colorizeExclusions():Array {return exclusions;}
  </fx:Script>

  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>

  <s:Button id="incrementButton" right="0" top="0" height="50%"
    skinClass="mySkins.SpinnerUpButton" />
  <s:Button id="decrementButton" right="0" bottom="0" height="50%"
    skinClass="mySkins.SpinnerDownButton" />

  <s:TextInput id="textDisplay" left="0" top="0" right="18" bottom="0"
    skinClass="spark.skins.spark.NumericStepperTextInputSkin" />

</s:SparkSkin>

```

The following skin classes replace the `SpinnerIncrementButtonSkin` and `SpinnerDecrementButtonSkin` classes in the custom `NumericStepperSkin` example. Instead of drawing arrows, these custom skins use + and - signs for incrementing and decrementing the value of the `TextInput` control.

```

<codeblock> [Outputclass: NoSWF]
Resolved code-reference.
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/SpinnerDownButton.mxml -->
<s:SparkSkin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

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

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

  <!-- border/fill -->
  <s:Path data="M 0 0 h 18 v 8 Q 18 9 16 10 h -16 Z"
    left="0" top="0" right="0" bottom="0">
    <s:stroke>
      <s:SolidColorStroke color="0x686868" weight="1"/>
    </s:stroke>
    <s:fill>
      <s:LinearGradient rotation="90">
        <s:GradientEntry color="0xE8E8E8"
          color.over="0xC2C2C2"
          color.down="0xAEB0B1" />
        <s:GradientEntry color="0xDFDFDF"
          color.over="0xADAEAF"
          color.down="0xA1A3A5" />
      </s:LinearGradient>
    </s:fill>
  </s:Path>

  <!-- highlight -->
  <s:Path data="M 0 0 h 16 v 6 Q 16 8 14 8 h -14 Z"
    left="1" top="1" right="1" bottom="1" >
    <s:stroke>
      <s:LinearGradientStroke rotation="90" weight="1">
        <s:GradientEntry color="0xFFFFFFFF"
          color.down="0x000000"
          alpha="0.55"
          alpha.over="0.55"
          alpha.down="0.15" />
        <s:GradientEntry color="0xFFFFFFFF"
          color.down="0x000000"
          alpha="0.2475"

```

```

                alpha.over="0.2475"
                alpha.down="0" />
            </s:LinearGradientStroke>
        </s:stroke>
    </s:Path>

    <!-- shadow -->
    <s:Rect left="1" top="2" right="1" height="1" includeIn="down">
        <s:fill>
            <s:SolidColor color="0x000000" alpha="0.07" />
        </s:fill>
    </s:Rect>

    <!-- Replace the down arrow with a minus sign. -->
    <s:Label id="arrow"
        text="-"
        horizontalCenter="0"
        verticalCenter="0">
    </s:Label>
</s:SparkSkin>

```

<codeblock> [Outputclass: NoSWF]

Resolved code-reference.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/SpinnerUpButton.mxml -->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

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

    <fx:Script>
    </fx:Script>

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

    <!-- border/fill -->
    <s:Path data="M 0 0 h 18 v 8 Q 18 9 16 10 h -16 Z"
        left="0" top="0" right="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0x686868" weight="1"/>
        </s:stroke>
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0xE8E8E8"
                    color.over="0xC2C2C2"
                    color.down="0xAEB0B1" />
                <s:GradientEntry color="0xDFDFDF"
                    color.over="0xADAEAF"

```

```

                color.down="0xA1A3A5" />
            </s:LinearGradient>
        </s:fill>
    </s:Path>

    <!-- highlight -->
    <s:Path data="M 0 0 h 16 v 6 Q 16 8 14 8 h -14 Z"
        left="1" top="1" right="1" bottom="1" >
        <s:stroke>
            <s:LinearGradientStroke rotation="90" weight="1">
                <s:GradientEntry color="0xFFFFFFFF"
                    color.down="0x000000"
                    alpha="0.55"
                    alpha.over="0.55"
                    alpha.down="0.15" />
                <s:GradientEntry color="0xFFFFFFFF"
                    color.down="0x000000"
                    alpha="0.2475"
                    alpha.over="0.2475"
                    alpha.down="0" />
            </s:LinearGradientStroke>
        </s:stroke>
    </s:Path>

    <!-- shadow -->
    <s:Rect left="1" top="2" right="1" height="1" includeIn="down">
        <s:fill>
            <s:SolidColor color="0x000000" alpha="0.07" />
        </s:fill>
    </s:Rect>

    <!-- Replace the up arrow with a plus sign. -->
    <s:Label id="arrow"
        text="+"
        horizontalCenter="0"
        verticalCenter="0">
    </s:Label>
</s:SparkSkin>

```

Packaging skins

[Output: IPH, Print, Web] [Revision Control: Changing]

You can package custom Spark skins as a SWC file and distribute that SWC file to anyone interested in using your library of skins. The SWC file is also known as a theme SWC file. Theme SWC files for Spark skins typically consist of the following files:

- One or more CSS files
- One or more skin classes

The CSS files in the theme SWC file can apply the Spark skin classes and any other style properties. The following sample CSS file applies a style property and custom skins to the Button and CheckBox controls in the Spark namespace:

```
@namespace s "library://ns.adobe.com/flex/spark";
s|Button {
    color: Green;
    skinClass: ClassReference("ButtonTransitionSkin");
}
s|CheckBox {
    color: Green;
    skinClass: ClassReference("CheckBoxTransitionSkin");
}
```

To include classes that are compiled, such as Spark skins, in a theme SWC file, you use the `include-classes` compiler option. To include files that are not compiled, such as a stylesheet, you use the `include-file` compiler option. All files included in a theme SWC file must be in the source path. You use the `output` compiler option to specify the location of the resulting SWC file.

The following command line compiles a new theme SWC file. This theme SWC file includes two Spark skins and a CSS file:

```
compc -source-path c:/temp/myskins
      -include-classes ButtonTransitionSkin CheckBoxTransitionSkin
      -include-file transition.css c:/temp/myskins/transition.css
      -output c:/temp/myskins/MySkins.swc
```

To use the theme SWC file in your application, you use the `theme` compiler option, as the following example shows:

```
mxmhc -theme=c:/temp/myskins/MySkins.swc c:myapps/ThemeExample.xml
```

For more information, see [Creating a theme SWC file](#) and [Using compc](#), the component compiler.