

Chapter 1: Introduction to effects

Effects let you add animation to your application in response to user or programmatic action. For example, you can use effects to cause a dialog box to bounce slightly when it receives focus, or to slowly fade in when it becomes visible. You build effects into your applications by using MXML and ActionScript.

About effects

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

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

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

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

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

About applying Spark effects

Spark effects are defined in the `spark.effects` package. To apply a Spark effect, you first define it in the `<fx:Declarations>`, and then invoke the effect by calling the `Effect.play()` method. The following example uses the event listener of a Button control's `click` event to invoke a `Resize` effect on an Image control:

```
<?xml version="1.0"?>
<!-- behaviorExamples\SparkResizeEffect.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <s:Resize id="myResizeEffect"
      target="{myImage}"
      widthBy="10" heightBy="10"/>
  </fx:Declarations>

  <mx:Image id="myImage"
    source="@Embed(source='assets/logo.jpg')"/>
  <s:Button label="Resize Me"
    click="myResizeEffect.end();myResizeEffect.play();"/>
</s:Application>
```

Notice that this example first calls the `Effect.end()` method before it calls the `play()` method. Call the `end()` method to ensure that any previous instance of the effect has ended before you start a new one.

In the next example, you create two `Resize` effects for a `Button` control. One `Resize` effect expands the size of the button by 10 pixels when the user clicks down on the button, and the second resizes it back to its original size when the user releases the mouse button. The duration of each effect is 200 ms:

```
<?xml version="1.0"?>
<!-- behaviorExamples\SparkResizeEffectReverse.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <s:Resize id="myResizeEffectUp"
      target="{myImage}"
      widthBy="10" heightBy="10"/>
    <s:Resize id="myResizeEffectDown"
      target="{myImage}"
      widthBy="-10" heightBy="-10"/>
  </fx:Declarations>

  <mx:Image id="myImage"
    source="@Embed(source='assets/logo.jpg')"/>

  <s:Button label="Resize Me Up"
    click="myResizeEffectUp.end();myResizeEffectUp.play();"/>

  <s:Button label="Resize Me Down"
    click="myResizeEffectDown.end();myResizeEffectDown.play();"/>
</s:Application>
```

About applying Halo effects

Halo effects are defined in the `mx.effects` package. With Halo effects, you typically use a trigger to initiate the effect. A *trigger* is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. To configure a component to use an effect, you associate an effect with a trigger.

Note: *Triggers are not the same as events. For example, a `Button` control has both a `mouseDown` event and a `mouseDownEffect` trigger. The event initiates the corresponding effect trigger when a user clicks on a component. You use the `mouseDown` event property to specify the event listener that executes when the user clicks on the component. You use the `mouseDownEffect` trigger property to associate an effect with the trigger.*

You associate Halo effects with triggers as part of defining the basic behavior for your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWL.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <!-- Define effect. -->
    <mx:WipeLeft id="myWL" duration="1000"/>
  </fx:Declarations>

  <!-- Assign effect to targets. -->
  <mx:Button id="myButton" label="Click Me"
    mouseDownEffect="{myWL}"/>
  <mx:Button id="myOtherButton" label="Click Me"
    mouseDownEffect="{myWL}"/>
</s:Application>
```

In this example, the effect is a `WipeLeft` effect with a duration of 1000 milliseconds (ms). That means it takes 1000 ms for the effect to play from start to finish.

You use data binding to assign the effect to the `mouseDownEffect` property of each `Button` control. The `mouseDownEffect` property is the effect trigger that specifies to play the effect when the user clicks the control using the mouse pointer. In the previous example, the effect makes the `Button` control appear as if it is being wiped onto the screen from right to left.

About factory and instance classes

Flex implements effects using an architecture in which each effect is represented by two classes: a factory class and an instance class.

Factory class You use factory classes in your application. The factory class creates an object of the instance class to perform the effect on the target. The factory classes are defined in the `spark.effects` and `mx.effects` packages.

You define a factory class in your application, and configure it with the necessary properties to control the effect, such as the zoom size or effect duration, as the following example shows:

```
<fx:Declarations>
  <!-- Factory class. -->
  <s:Resize id="myResizeEffect"
    target="{myImage}"
    widthBy="50" heightBy="50"/>
</fx:Declarations>
<!-- Effect target.-->
<mx:Image id="myImage"/>
<s:Button label="Resize Image"
  click="myResizeEffect.end();myResizeEffect.play();"/>
```

By convention, the name of a factory class is the name of the effect, such as `Resize`, `Move`, or `Fade`.

Instance class The instance class implements the effect logic. When an effect trigger occurs, or when you call the `play()` method to invoke an effect, the factory class creates an object of the instance class to perform the effect on the target. When the effect ends, Flex destroys the instance object. If the effect has multiple target components, the factory class creates multiple instance objects, one per target.

The factory classes are defined in the `spark.effects.supportClasses` and `mx.effects.effectClasses` packages. By convention, the name of an instance class is the name of the factory class with the suffix *Instance*, such as `ResizeInstance`, `MoveInstance`, or `FadeInstance`.

When you use effects, you perform the following steps.

- 1 Create a factory class object in your application.
- 2 Configure the factory class object.

When Flex plays an effect, Flex performs the following actions:

- 1 Creates one object of the instance class for each target component of the effect.
- 2 Copies configuration information from the factory object to the instance object.
- 3 Plays the effect on the target using the instance object.
- 4 Deletes the instance object when the effect completes.

Any changes that you make to the factory object are not propagated to a currently playing instance object. However, the next time the effect plays, the instance object uses your new settings.

When you use effects in your application, you are concerned only with the factory class; the instance class is an implementation detail. However, if you want to create custom effects classes, you must implement a factory and an instance class.

For more information, see [Custom Effects](#).

Available effects

The following table lists the Spark and Halo effects:

Spark Effect	Halo Effect	Description
Animate Animate3D AnimateColor	AnimateProperty	<p>Animates a numeric property of a component, such as <code>height</code>, <code>width</code>, <code>scaleX</code>, or <code>scaleY</code>. You specify the property name, start value, and end value of the property to animate. The effect first sets the property to the start value, and then updates the property value over the duration of the effect until it reaches the end value.</p> <p>For example, if you want to change the width of a Button control, you can specify <code>width</code> as the property to animate, and starting and ending width values to the effect.</p>
AnimateFilter		Animates the properties of a filter applied to the target. This effect does not actually modify the properties of the target. For example, you can use this property to animate a filter, such as a <code>DropShadowFilter</code> , applied to the target.
Move	Move	Changes the position of a component over a specified time interval. You typically apply this effect to a target in a container that uses absolute positioning, such as a Halo Canvas container, or a Spark container that uses <code>BasicLayout</code> . If you apply it to a target in a container that performs automatic layout, the move occurs, but the next time the container updates its layout, it moves the target back to its original position. You can set the container's <code>autoLayout</code> property to <code>false</code> to disable the move back, but that disables layout for all controls in the container.
Rotate Rotate3D	Rotate	<p>Rotates a component around a specified point. You can specify the coordinates of the center of the rotation, and the starting and ending angles of rotation. You can specify positive or negative values for the angles.</p> <p>Note: To use the rotate effects with Halo components that display text, the component must either support the Flash Text Engine or you must use an embedded font with the component, not a device font. For more information, see <i>Using Styles and Themes</i>.</p>
ScaleScale3D		Scale a component. You can specify properties to scale the target in the x and y directions.
	Blur	<p>Applies a blur visual effect to a component. A Blur effect softens the details of an image. You can produce blurs that range from a softly unfocused look to a Gaussian blur, a hazy appearance like viewing an image through semi-opaque glass. If you apply a Blur effect to a component, you cannot apply a <code>BlurFilter</code> or a second Blur effect to the component.</p> <p>The Blur effect uses the Flash <code>BlurFilter</code> class as part of its implementation. For more information, see <code>flash.filters.BlurFilter</code> in the <i>Adobe Flex Language Reference</i>.</p>
CallAction		Animates the target by calling a function on the target. The effect lets you pass parameters to the function from the effect class.
	Dissolve	<p>Modifies the <code>alpha</code> property of an overlay to gradually have to target component appear or disappear. If the target object is a container, only the children of the container dissolve. The container borders do not dissolve.</p> <p>Note: To use the Halo Dissolve effect with the <code>creationCompleteEffect</code> trigger of a <code>DataGrid</code> control, you must define the data provider of the control inline using a child tag of the <code>DataGrid</code> control, or using data binding. This issue is a result of the data provider not being set until the <code>creationComplete</code> event is dispatched. Therefore, when the effect starts playing, Flex has not completed the sizing of the <code>DataGrid</code> control.</p>

Spark Effect	Halo Effect	Description
Fade CrossFade	Fade	<p>Animate the component from transparent to opaque, or from opaque to transparent.</p> <p>If you specify the Halo Fade effect for the <code>showEffect</code> or <code>hideEffect</code> trigger, and if you omit the <code>alphaFrom</code> and <code>alphaTo</code> properties, the effect automatically transitions from 0.0 to the targets' current <code>alpha</code> value for a show trigger, and from the targets' current <code>alpha</code> value to 0.0 for a hide trigger.</p> <p>Note: To use these effects with Halo components that display text, the component must either support the Flash Text Engine or you must use an embedded font with the component, not a device font. For more information, see Using Styles and Themes.</p>
	Glow	<p>Applies a glow visual effect to a component. The Glow effect uses the Flash <code>GlowFilter</code> class as part of its implementation. For more information, see the <code>flash.filters.GlowFilter</code> class in the <i>Adobe Flex Language Reference</i>. If you apply a Glow effect to a component, you cannot apply a <code>GlowFilter</code> or a second Glow effect to the component.</p>
	Iris	<p>Animates the effect target by expanding or contracting a rectangular mask centered on the target. The effect can either grow the mask from the center of the target to expose the target, or contract it toward the target center to obscure the component.</p> <p>For more information, see Using a mask effect.</p>
	Pause	<p>Does nothing for a specified period of time. This effect is useful when you need to composite effects. For more information, see "Creating composite effects" on page 16.</p>
Resize	Resize	<p>Changes the width and height of a component over a specified time interval. When you apply a Resize effect, the layout manager resizes neighboring components based on the size changes to the target component. To run the effect without resizing other components, place the target component in a Canvas container, or a Spark container that uses <code>BasicLayout</code>.</p> <p>When you use the Resize effect with Panel containers, you can hide Panel children to improve performance. For more information, see Improving performance when resizing Panel containers.</p>
	SoundEffect	<p>Plays an MP3 audio file. For example, you could play a sound when a user clicks a Button control. This effect lets you repeat the sound, select the source file, and control the volume and pan.</p> <p>You specify the MP3 file using the <code>source</code> property. If you have already embedded the MP3 file, using the <code>Embed</code> keyword, then you can pass the Class object of the MP3 file to the <code>source</code> property. Otherwise, specify the full URL to the MP3 file.</p> <p>For more information, see Using a sound effect.</p>
Wipe	WipeLeft WipeRight WipeUp WipeDown	<p>Defines a bar Wipe effect. The before or after state of the component must be invisible.</p>
	Zoom	<p>Zooms a component in or out from its center point by scaling the component.</p> <p>Note: When you apply a Zoom effect to text rendered using a system font, Flex scales the text between whole point sizes. Although you do not have to use embedded fonts when you apply a Zoom effect to text, the Zoom will appear smoother when you apply it to embedded fonts. For more information, see Using Styles and Themes.</p>

Applying effects

To apply an effect to a target, you must specify the target, and then initiate the effect. Once the effect has started, you can pause, stop, resume, and end the effect epigrammatically.

Using the `Effect.target` and `Effect.targets` properties

Use the `Effect.target` or `Effect.targets` properties to specify the effect targets. You use the `Effect.target` property in MXML to specify a single target, and the `Effect.targets` property to specify an array of targets, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\TargetProp.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:Declarations>
    <s:Resize id="myResize"
      heightBy="25"
      widthBy="50"
      target="{myButton}"/>
  </fx:Declarations>

  <s:Button id="myButton"
    label="Resize target"
    click="myResize.end();myResize.play();"/>
</s:Application>
```

In this example, you use data binding to the `target` property to specify that the `Button` control is the target of the `Resize` effect.

In the next example, you apply a `Resize` effect to multiple `Button` controls by using data binding with the effect's `targets` property:

```
<?xml version="1.0"?>
<!-- behaviors\TargetProp3Buttons.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <s:Resize id="myResize"
      heightBy="25"
      widthBy="50"
      targets="{[myButton1, myButton2, myButton3]}" />
  </fx:Declarations>

  <s:Button id="myButton1" label="Button 1" />
  <s:Button id="myButton2" label="Button 2" />
  <s:Button id="myButton3" label="Button 3" />

  <s:Button id="myButton4"
    label="Zoom targets"
    click="myResize.end();myResize.play();" />
</s:Application>
```

Because you specified three targets to the effect, the `play()` method invokes the effect on all three button controls.

If you use the `targets` property to define multiple event targets, calling the `end()` method with no arguments terminates the effect on all targets. If you pass an effect instance as an argument, just that instance is interrupted.

To obtain the effect instance, save the return value of the `play()` method, as the following example shows:

```
var myResizeArray:Array = myResize.play();
```

The Array contains `EffectInstance` objects, one per target of the effect. Pass the element from the Array to the `end()` method that corresponds to the effect to end, as the following example shows:

```
myResize.end(myResizeArray[1]);
```

You can define an effect that specifies no targets. Instead, you can pass an Array of targets to the `play()` method to invoke the effect on all components specified in the Array, as the following example shows:

```
myResize.play([comp1, comp2, comp3]);
```

This example invokes the `Resize` effect on three components.

Applying effects by using data binding

You can use data binding in MXML to set properties of an effect. For example, the following example lets the user set the `heightBy` and `widthBy` properties of the `Resize` effect using a `TextInput` control. The `heightBy` and `widthBy` properties specify the increase, of pixels, of the size of the target.

```

<?xml version="1.0"?>
<!-- behaviors\DatabindingEffects.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <s:Resize id="myResizeEffect"
      target="{myImage}"
      heightBy="{Number(resizeHeightInput.text)}"
      widthBy="{Number(resizeWidthInput.text) }"/>
  </fx:Declarations>

  <mx:Form>
    <mx:FormItem label="Resize Height:">
      <s:TextInput id="resizeHeightInput"
        text="0" width="30"/>
    </mx:FormItem>
    <mx:FormItem label="Resize Width:">
      <s:TextInput id="resizeWidthInput"
        text="0" width="30"/>
    </mx:FormItem>
  </mx:Form>

  <mx:Image id="myImage"
    maintainAspectRatio="false"
    source="@Embed(source='assets/logo.jpg')"/>
  <s:Button label="Resize Image" click="myResizeEffect.play();" />
</s:Application>

```

Playing an effect backward

You can pass an optional argument to the `play()` method to play the effect backward, as the following example shows:

```
resizeLarge.play([comp1, comp2, comp3], true);
```

In this example, you specify `true` as the second argument to play the effect backward. The default value is `false`.

You can also use the `Effect.pause()` method to pause an effect, the `Effect.resume()` method to resume a paused effect, and the `Effect.reverse()` method to play an effect backward.

Ending an effect

Use the `end()` method to terminate an effect at any time, as the following example shows:

```

<?xml version="1.0"?>
<!-- behaviors\ASend.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:Declarations>
    <s:Resize id="resizeLarge"
      heightTo="150"
      widthTo="150"
      duration="5000"
      target="{myTA}"/>
  </fx:Declarations>

  <s:Group height="228" width="328">
    <s:Button label="Start"
      x="10" y="10"
      click="resizeLarge.end();resizeLarge.play();"/>
    <s:Button label="End"
      x="86" y="10"
      click="resizeLarge.end();"/>
    <s:Button label="Reset"
      click="myTA.height=100;myTA.width=100;"
      x="162" y="10"/>
    <s:TextArea id="myTA"
      x="10" y="40"
      height="100"
      width="100"
      text="Here is some text."/>
  </s:Group>
</s:Application>

```

In this example, you set the `duration` property of the `Resize` effect to 10 seconds, and use a `Button` control to call the `end()` method to terminate the effect when the user clicks the button.

When you call the `end()` method, the effect jumps to its end state and then terminates. In the case of the `Resize` effect, the effect sets the final size of the expanded `TextArea` control before it terminates, just as if you had let the effect finish playing. If the effect was a `move` effect, the target component moves to its final position before terminating.

You can end all effects on a component by calling the `UIComponent.endEffectsStarted()` method on the component. The `endEffectsStarted()` method calls the `end()` method on every effect currently playing on the component.

If you defined a listener for the `effectEnd` event, that listener gets invoked by the `end()` method, just as if you had let the effect finish playing. For more information on working with effect events, see “Handling effect events” on page 13.

After the effect starts, you can use the `pause()` method to pause the effect at its current location. You can then call the `resume()` method to start the effect, or the `end()` method to terminate it.

Use the `stop()` method to halt the effect in its current state, but not jump to the end. A call to the `stop()` method dispatches the `effectEnd` event. Unlike a call to the `pause()` method, you cannot call the `resume()` method after calling the `stop()` method. However, you can call the `play()` method to restart the effect.

Creating effects in ActionScript

You can declare and play effects in ActionScript. The following example uses the event listener of a `Button` control’s `click` event to invoke a `Resize` effect on an `TextArea` control:

```

<?xml version="1.0"?>
<!-- behaviors\ASplayVBox.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"
  creationComplete="createEffect(event);">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    <![CDATA[
      // Import effect class.
      import spark.effects.Resize;

      // Create a resize effect
      private var resizeLarge:Resize = new Resize();

      private function createEffect(eventObj:Event):void {
        // Set the TextArea as the effect target.
        resizeLarge.target=myTA;

        // Set resized width and height, and effect duration.
        resizeLarge.widthTo=150;
        resizeLarge.heightTo=150;
        resizeLarge.duration=750;
      }
    ]]>
  </fx:Script>

  <s:Button label="Start"
    click="resizeLarge.end();resizeLarge.play();"/>
  <s:Button label="Reset"
    click="myTA.width=100;myTA.height=100;"/>
  <s:TextArea id="myTA"
    height="100" width="100"
    text="Here is some text."/>
</s:Application>

```

In this example, use the application's `creationComplete` event to configure the effect, and then invoke it by calling the `play()` method in response to a user clicking the Button control.

Working with effects

Effects have many configuration settings that you can use to control the effect. For example, you can set the effect duration and repeat behavior, or handle effect events. The effect target also has configuration settings that you can use to configure it for effects.

Setting effect durations

All effects take the `duration` property that you can use to specify the time, in milliseconds, over which the effect occurs. The following example creates two versions of the Fade effect. The `slowFade` effect uses a two-second duration; the `reallySlowFade` effect uses an eight-second duration:

```

<?xml version="1.0"?>
<!-- behaviors\FadeDuration.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <s:Fade id="slowFade"
      duration="2000"
      target="{myButton1}"/>
    <s:Fade id="reallySlowFade"
      duration="8000"
      target="{myButton2}"/>
  </fx:Declarations>

  <s:Button id="myButton1"
    label="Button 1"
    creationCompleteEffect="{slowFade}"/>
  <s:Button id="myButton2"
    label="Button 2"
    creationCompleteEffect="{reallySlowFade}"/>
</s:Application>

```

Delaying effect start

The `Effect.startDelay` property specifies a value, in milliseconds, that the effect waits once it is triggered before it begins. You can specify an integer value greater than or equal to 0. If you have used the `Effect.repeatCount` property to specify the number of times to repeat the effect, the `startDelay` property is applied only to the first time the effect plays, but not to the repeated playing of the effect.

If you set the `startDelay` property for a Parallel effect, Flex inserts the delay between each effect of the parallel effect.

Repeating effects

All effects support the `Effect.repeatCount` and `Effect.repeatDelay` properties that let you configure whether effects repeat, where:

- `repeatCount` Specifies the number of times to play the effect. A value of 0 means to play the effect indefinitely until stopped by a call to the `end()` method. The default value is 1. For a repeated effect, the `duration` property specifies the duration of a single instance of the effect. Therefore, if an effect has a `duration` property set to 2000, and a `repeatCount` property set to 3, then the effect takes a total of 6000 ms (6 seconds) to play.
- `repeatDelay` Specifies the amount of time, in milliseconds, to pause before repeating the effect. The default value is 0.
- `repeatBehavior` (Spark effects only) Specifies `RepeatBehavior.LOOP` (default) to repeat the effect each time, or `RepeatBehavior.REVERSE` to reverse direction of the effect on each iteration.

For example, the following example repeats the Rotate effect until the user clicks a Button control:

```

<?xml version="1.0"?>
<!-- behaviors\RepeatEff.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <s:Rotate id="myRotate"
      angleBy="360"
      repeatCount="0"
      target="{myImage}"/>
  </fx:Declarations>

  <s:SimpleText text="Click the image to start rotation."/>
  <s:Button id="myButton"
    label="Stop Rotation"
    click="myRotate.end();"/>
  <mx:Image id="myImage"
    source="@Embed(source='assets/logo.jpg')"
    mouseDown="myRotate.end(); myRotate.play();"/>
</s:Application>

```

All effects dispatch an `effectEnd` event when the effect completes. If you repeat the effect, the effect dispatches the `effectEnd` event after the final repetition.

If the effect is a tween effect, such as a Halo Fade or Halo Move effect, the effect dispatches both the `tweenEnd` effect and the `endEffect` when the effect completes. If you configure the tween effect to repeat, the `tweenEnd` effect occurs at the end of every repetition of the effect, and the `endEffect` event occurs after the final repetition.

Handling effect events

Every Spark and Halo effect class supports the following events:

- `effectStart` Dispatched when the effect starts playing. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_START`.
- `effectEnd` Dispatched after the effect ends, either when the effect finishes playing or when the effect has been interrupted by a call to the `end()` method. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_END`.
- `effectStop` Dispatched after the effect stops by a call to the `stop()` method. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_STOP`.

The event object passed to the event listener for these events is of type `EffectEvent`.

The previous list of events are dispatched by all effects. The Spark and Halo effects dispatch additional events. For more information, see [Handling Spark effect events](#) and [Handling Halo effect events](#).

Flex dispatches one event for each target of an effect. Therefore, if you define a single target for an effect, Flex dispatches a single `effectStart` event, and a single `effectEnd` event. If you define three targets for an effect, Flex dispatches three `effectStart` events, and three `effectEnd` events.

The `EffectEvent` class is a subclass of the `Event` class, and contains all of the properties inherited from `Event`, including `target`, and `type`, and defines a new property named `effectInstance`, where:

target Contains a reference to the `Effect` object that dispatched the event. This is the factory class of the effect.

type Either `EffectEvent.EFFECT_END` or `EffectEvent.EFFECT_START`, depending on the event.

effectInstance Contains a reference to the `EffectInstance` object. This is the object defined by the instance class for the effect. Flex creates one object of the instance class for each target of the effect. You access the target component of the effect using the `effectInstance.target` property.

The following example defines an event listener for the `endEffect` event:

```
<?xml version="1.0"?>
<!-- behaviors\EventEffects2.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:VerticalLayout/>
  </s:layout>

  <fx:Script>
    <![CDATA[

      import mx.effects.*;
      import mx.events.EffectEvent;
      import mx.core.UIComponent;

      private function endEffectListener(eventObj:EffectEvent):void {
        // Access the effect object.
        var effectObj:Effect = Effect(eventObj.target);

        // Access the target component of the effect.
        var effectTarget:UIComponent =
          UIComponent(eventObj.effectInstance.target);
        // Write the target id and event type to the TextArea control.
        myTA.text = effectTarget.id;
        myTA.text = myTA.text + " " + eventObj.type;
      }
    ]]>
  </fx:Script>

  <fx:Declarations>
    <s:Fade id="slowFade"
      duration="2000"
      effectEnd="endEffectListener(event);"
      target="{myButton1}"/>
  </fx:Declarations>

  <s:Button id="myButton1"
    label="Button 1"
    creationCompleteEffect="{slowFade}"/>

  <s:TextArea id="myTA" />
</s:Application>
```

If the effect has multiple targets, Flex dispatches an `effectStart` event and `effectEnd` event once per target, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\EventEffects.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:VerticalLayout/>
  </s:layout>

  <fx:Script>
    <![CDATA[

      import mx.effects.*;
      import mx.events.EffectEvent;
      import mx.core.UIComponent;

      private function endSlowFadeEffectListener(eventObj:EffectEvent):void
      {
        // Access the effect object.
        var effectObj:Effect = Effect(eventObj.target);

        // Access the target component of the effect.
        var effectTarget:UIComponent =
          UIComponent(eventObj.effectInstance.target);

        myTA.text = myTA.text + effectTarget.id + ' : ';
        myTA.text = myTA.text + " " + eventObj.type + '\n';
      }
    ]]>
  </fx:Script>

  <fx:Declarations>
    <s:Fade id="slowFade"
      duration="2000"
      effectEnd="endSlowFadeEffectListener(event);"
      targets="{[myButton1, myButton2, myButton3, myButton4]}" />
  </fx:Declarations>

  <s:Button id="myButton1"
    label="Button 1"
    creationCompleteEffect="{slowFade}" />
  <s:Button id="myButton2"
    label="Button 2"
    creationCompleteEffect="{slowFade}" />
  <s:Button id="myButton3"
    label="Button 3"
    creationCompleteEffect="{slowFade}" />
  <s:Button id="myButton4"
    label="Button 4"
    creationCompleteEffect="{slowFade}" />

  <s:TextArea id="myTA" height="125" width="250" />
</s:Application>
```

Flex dispatches an `effectEnd` event once per target; therefore, the `endSlowFadeEffectListener()` event listener is invoked four times, once per Button control.

Creating composite effects

Flex supports two ways to combine, or *composite*, effects:

Parallel The effects play at the same time. If you play effects in parallel, you must make sure that the effects do not modify the same property of the target. If two effects modify the same property, the effects conflict with each other and the results of the effects are undefined.

Sequence One effect must complete before the next effect starts.

To define a Parallel or Sequence effect, you use the `<Parallel>` or `<Sequence>` tag. The following example defines the Parallel effect, `fadeResizeShow`, which combines the Spark Fade and Resize effects in parallel, and `fadeResizeHide`, which combines the Fade and Resize effects in sequence:

```
<?xml version="1.0"?>
<!-- behaviors\CompositeEffects.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:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <s:Sequence id="fadeResizeHide"
      target="{aTextArea}"
      duration="1000">
      <s:Fade id="fadeHide"
        alphaFrom="1.0"
        alphaTo="0.0"/>
      <s:Resize id="resizeHide"
        widthTo="0"
        heightTo="0"/>
    </s:Sequence>
    <s:Parallel id="fadeResizeShow"
      target="{aTextArea}"
      duration="1000">
```

```

        <s:Resize id="resizeShow"
            widthTo="100"
            heightTo="50"/>
        <s:Fade id="fadeShow"
            alphaFrom="0.0"
            alphaTo="1.0"/>
    </s:Parallel>
</fx:Declarations>

<s:TextArea id="aTextArea"
    width="100" height="50"
    text="Hello world."/>

<s:Button id="myButton2"
    label="Hide!"
    click="fadeResizeHide.end();fadeResizeHide.play();"/>
<s:Button id="myButton1"
    label="Show!"
    click="fadeResizeShow.end();fadeResizeShow.play();"/>
</s:Application>

```

The button controls alternates making the TextArea control visible and invisible. When the TextArea control becomes invisible, it uses the fadeResizeHide effect as its hide effect, and when it becomes visible, it uses the fadeResizeShow effect.

As a modification to this example, you could disable the Show button when the TextArea control is visible, and disable the Hide button when the TextArea control is invisible, as the following example shows:

```

<?xml version="1.0"?>
<!-- behaviors\CompositeEffectsEnable.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:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            private function showHandler():void {
                myButton2.enabled=true;
                myButton1.enabled=false;
            }

            private function hideHandler():void {
                myButton2.enabled=false;
                myButton1.enabled=true;
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:Sequence id="fadeResizeHide"
            target="{aTextArea}"
            duration="1000"
            effectEnd="hideHandler();">
            <s:Fade id="fadeHide"

```

```

        alphaFrom="1.0"
        alphaTo="0.0"/>
    <s:Resize id="resizeHide"
        widthTo="0"
        heightTo="0"/>
</s:Sequence>
<s:Parallel id="fadeResizeShow"
    target="{aTextArea}"
    duration="1000"
    effectEnd="showHandler();">
    <s:Resize id="resizeShow"
        widthTo="100"
        heightTo="50"/>
    <s:Fade id="fadeShow"
        alphaFrom="0.0"
        alphaTo="1.0"/>
</s:Parallel>
</fx:Declarations>

<s:TextArea id="aTextArea"
    width="100" height="50"
    text="Hello world."/>

<s:Button id="myButton2"
    label="Hide!"
    click="fadeResizeHide.end();fadeResizeHide.play();"/>
<s:Button id="myButton1"
    label="Show!"
    enabled="false"
    click="fadeResizeShow.end();fadeResizeShow.play();"/>
</s:Application>

```

In this example, the Show button is initially disabled. Event handlers for the `effectEnd` event toggle the `enable` property for the two button based on the effect that played.

You can nest `<Parallel>` and `<Sequence>` tags inside each other. For example, two effects can run in parallel, followed by a third effect running in sequence.

In a Parallel or Sequence effect, the `duration` property sets the duration of each effect. For example, if the a Sequence effect has its `duration` property set to 3000, then each effect in the Sequence will take 3000 ms to play.

Using embedded fonts with effects

The fade and rotate effects only work with components that support the Flash Text Engine (FTE), or with components that use an embedded font. All Spark components, and some Halo components, support the FTE. Therefore, the fade and rotate effects work with text in the components.

However, if you apply a fade and rotate effects to a Halo component that uses a system font, nothing happens to the text in the component. You either have to embed a font, or use the appropriate Spark component instead of the Halo component.

When you apply a Halo Zoom effect to text rendered using a system font, Flex scales the text between whole point sizes. While you do not have to use embedded fonts when you apply a Zoom effect to text, the Zoom will appear smoother when you apply it to embedded fonts.

The following example rotates text area controls. The first text area control is defined using the Spark `TextArea` component, which supports the FTE. Therefore the text rotates when you apply the Spark Rotate effect to it.

The second text area control is defined using the Halo TextArea control. This TextArea control uses an embedded font and, therefore, the text rotates.

The third text area control uses the Halo TextArea control with the default system font. Therefore, when you apply the Spark Rotate effect, the text disappears and reappears when you rotate the component back to its initial state:

```
<?xml version="1.0"?>
<!-- behaviors\EmbedFont.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>

  <fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    @namespace mx "library://ns.adobe.com/flex/halo";
    @font-face {
      src:url("../assets/MyriadWebPro.ttf");
      font-family: myMyriadWebPro;
    }
  </fx:Style>

  <fx:Declarations>
    <s:Rotate id="rotateForward"
      angleFrom="0" angleTo="45"/>
    <s:Rotate id="rotateBack"
      angleFrom="45" angleTo="0"/>
  </fx:Declarations>

  <s:VGroup>
    <s:Button label="Rotate Forward"
      click="rotateForward.end();rotateForward.play([11]);"/>
    <s:Button label="Rotate Backward"
      click="rotateBack.end();rotateBack.play([11]);"/>
    <s:TextArea id="t1" height="75"
      fontFamily="myMyriadWebPro"
      text="FTE supported. This text will rotate."/>
  </s:VGroup>
```

```
<s:VGroup>
  <s:Button label="Rotate Forward"
    click="rotateForward.end();rotateForward.play([12]);"/>
  <s:Button label="Rotate Backward"
    click="rotateBack.end();rotateBack.play([12]);"/>
  <mx:TextArea id="12" height="75"
    fontFamily="myMyriadWebPro"
    text="Embedded font. This text will rotate."/>
</s:VGroup>

<s:VGroup>
  <s:Button label="Rotate Forward"
    click="rotateForward.end();rotateForward.play([13]);"/>
  <s:Button label="Rotate Backward"
    click="rotateBack.end();rotateBack.play([13]);"/>
  <mx:TextArea id="13" height="75"
    text="System font. This text will not rotate."/>
</s:VGroup>
</s:Application>
```

Suspending background processing

To improve the performance of effects, you can disable background processing in your application for the duration of the effect by setting the `Effect.suspendBackgroundProcessing` property to `true`. The background processing that is blocked includes component measurement and layout, and responses to data services for the duration of the effect.

The default value of the `suspendBackgroundProcessing` property is `false`. You can set it to `true` in most cases. However, you should set it to `false` if either of the following conditions is true for your application:

- User input may arrive while the effect is playing, and the application must respond to the user input before the effect finishes playing.
- A response may arrive from the server while the effect is playing, and the application must process the response while the effect is still playing.

Disabling container layout for effects

By default, Flex updates the layout of a container's children when a new child is added to it, when a child is removed from it, when a child is resized, and when a child is moved. Because some effects, such as the move and resize effects, modify a child's position or size, they cause the container to update its layout.

However, when the container updates its layout, it can actually reverse the results of the effect. For example, you use a move effect to reposition a container child. At some time later, you change the size of another container child, which forces the container to update its layout. This layout update can cause the child that moved to be returned to its original position.

To prevent Flex from performing layout updates, you can set the `autoLayout` property of a container to `false`. Its default value is `true`, which configures Flex so that it always updates layouts. You always set the `autoLayout` property on the parent container of the component that uses the effect. For example, if you want to control the layout of a child of a Grid container, you set the `autoLayout` property for the parent `GridItem` container of the child, not for the Grid container.

You set the `autoLayout` property to `false` when you use a move effect in parallel with a resize or zoom effect. You must do this because the resize or zoom effect can cause an update to the container's layout, which can return the child to its original location.

When you use the Zoom effect on its own, you can set the `autoLayout` property to `false`, or you may leave it with its default value of `true`. For example, if you use a Zoom effect with the `autoLayout` property set to `true`, as the child grows or shrinks, Flex automatically updates the layout of the container to reposition its children based on the new size of the child. If you use a Zoom effect with the `autoLayout` property set to `false`, the child resizes around its center point, and the remaining children do not change position.

The container in the following example uses the default vertical alignment of `top` and the default horizontal alignment of `left`. If you apply a Zoom effect to the image, the container resizes to hold the image, and the image remains aligned with the upper-left corner of the container:

```
<s:SkinnableContainer>
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <mx:Image source="myImage.jpg"/>
</s:SkinnableContainer>
```

In the next example, the image is centered in the container. If you apply a Zoom effect to the image, as it resizes, it remains centered in the container.

```
<s:SkinnableContainer>
  <s:layout>
    <s:VerticalLayout horizontalAlign="center"/>
  </s:layout>
  <mx:Image source="myImage.jpg"/>
</s:SkinnableContainer>
```

By default, the size of the container is big enough to hold the image at its original size. If you disable layout updates, and use the Zoom effect to enlarge the image, or use a move effect to reposition the image, the image might extend past the boundaries of the container, as the following example shows:

```
<s:SkinnableContainer autoLayout="false">
  <s:layout>
    <s:VerticalLayout horizontalAlign="center"/>
  </s:layout>
  <mx:Image source="myImage.jpg"/>
</s:SkinnableContainer>
```

For a Spark container, if you set the `autoLayout` property to `false`, the container does not resize as the image resizes. The image can grow to a size so that it extends beyond the boundaries of the container. You can then decide to wrap the container in the Scroller component to add scroll bars rather than allowing the Image to extend past the container boundaries.

For a Halo container, if you set the `autoLayout` property to `false`, the container does not resize as the image resizes. If the image grows to a size larger than the boundaries of the container, the container adds scroll bars and clips the image at its boundaries.

Setting `UIComponent.cachePolicy` on the effect target

An effect can use the bitmap caching feature in Adobe® Flash® Player to speed up animations. An effect typically uses bitmap caching when the target component's drawing does not change while the effect is playing.

For example, the Fade effect works by modifying the `alpha` property of the target component. Changing the `alpha` property does not change the way the target component is drawn on the screen. Therefore, caching the target component as a bitmap can speed up the performance of the effect. The Move effect modifies the `x` and `y` properties of the target component. Modifying the values of these properties does not alter the way the target component is drawn, so it can take advantage of bitmap caching.

Not all effects can use bitmap caching. Effects such as Zoom, resize, and the wipe effects modify the target component in a way that alters the way it is drawn on the screen. The Zoom effect modifies the scale properties of the component, which changes its size. Caching the target component as a bitmap for such an effect would be counterproductive because the bitmap changes continuously while the effect plays.

The `UIComponent.cachePolicy` property controls the caching operation of a component during an effect. The `cachePolicy` property can have the following values:

CachePolicy.ON Specifies that the effect target is always cached.

CachePolicy.OFF Specifies that the effect target is never cached.

CachePolicy.AUTO Specifies that Flex determines whether the effect target should be cached. This is the default value.

Flex uses the following rules to set the `cacheAsBitmap` property:

- When at least one effect that does not support bitmap caching is playing on a target, set the target's `cacheAsBitmap` property to `false`.
- When one or more effects that supports bitmap caching are playing on a target, set the target's `cacheAsBitmap` property to `true`.

Typically, you leave the `cachePolicy` property with its default value of `CachePolicy.AUTO`. However, you might want to set the property to `CachePolicy.OFF` because bitmap caching is interfering with your user interface, or because you know something about your application's behavior such that disabling bitmap caching will have a beneficial effect on it.