Digging Deeper into ActionScript Events[*]

R.G. (Dick) Baldwin

Abstract

Learn how to translate a largely Flex MXML program into a largely ActionScript program. Learn that events can be subdivided into two major categories: those that involve direct user interaction and those that don't. Learn how to use the events documentation to find what you need to write event-driven ActionScript programs.

NOTE: Click Effects03 [1] to run this ActionScript program. *(Click the "Back" button in your browser to return to this page.)*

1 Table of Contents

[*]Version 1.2: May 19, 2010 11:55 pm +0000
[†]http://creativecommons.org/licenses/by/3.0/
[1]http://cnx.org/content/m34452/latest/Effects03.html

2 Preface

2.1 General

NOTE: **ActionScript 3:** Note that all references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a continuing series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website* [2] *.)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website [3] in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

2.2 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.2.1 Figures

2.2.2 Listings

[2] http://www.dickbaldwin.com/tocFlex.htm
[3] http://www.dickbaldwin.com/tocFlex.htm

2.2.3 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [4] .

2.3 General background information

The ActionScript 3.0 Reference for the Adobe Flash Platform [5] is a voluminous document. Understanding ActionScript events on the basis of that document alone can be a daunting task.

There are two entry points into the documentation that can make it somewhat easier to navigate:

- The flash.events.Event [6] class
- The addEventListener [7] method

The flash.events.Event class

As I understand it, all possible ActionScript events are represented by subclasses of the flash.events.Event [8] class. That class has approximately eighty subclasses, many of which are further extended into additional subclasses.

The flash.events.MouseEvent class

For example, the flash.events.MouseEvent [9] class has eight subclasses, some of which are extended into other subclasses. Therefore, the total number of subclasses of the **Event** class may be well in excess of one hundred. This means that there may be more than one hundred different *types* of events being dispatched during the running of an ActionScript program.

Event types and subtypes

An ActionScript class represents a *type* . Therefore, each subclass of the **Event** class defines a different type of event. Many subclasses define a large number of subtypes.

For example, the **MouseEvent** class defines about seventeen subtypes [10] ranging in alphabetical order from **CLICK** to **ROLL_OVER** , including **MOUSE_MOVE** , **MOUSE_DOWN** , **MOUSE_UP** , etc.

(For brevity, I will refer to the subtypes as types for the remainder of this document.)

Hundreds of event types and hundreds of object types

Therefore, there are many hundreds of different types of events that can be dispatched in various combinations by hundreds of different types of objects during the running of an ActionScript program.

The DisplayObject class

[4] http://www.dickbaldwin.com/toc.htm
[5] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html
[6] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/events/Event.html
[7] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/events/EventDispatcher.html#addEventListener%28%29
[8] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/events/Event.html
[9] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/events/MouseEvent.html
[10] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/events/MouseEvent.html#top

Objects of some classes *(such as Button*[11] *)* , which are subclasses of the DisplayObject[12] class, can dispatch events as a result of direct user interaction. For example, a user can click on a button and cause a **click** event to be dispatched.

The URLLoader class

Objects of other classes *(such as URLLoader*[13] *)* , which are not subclasses of the DisplayObject[14] class, cannot be caused to dispatch an event as a result of direct user interaction. *(As far as I know, there is no way for a user to interact directly with an object of the* ***URLLoader*** *class.)*

However, an object of the **URLLoader** class can dispatch nine different types of events ranging in alphabetical order from **activate** to **securityError** .

The addEventListener methods

The documentation index[15] for Flex Builder 3 lists eight different versions of the **addEventListener** method in classes plus one that is declared in an interface. Of the eight versions defined in classes, one is defined in the EventDispatcher[16] class. That is the one that I will concentrate on in this lesson.

For the record, however, four of the eight versions are overridden versions of the method that is defined in the **EventDispatcher** class. The remaining three of the eight are defined in classes that extend the **Proxy** class, which I will ignore in this lesson.

The addEventListener method in the EventDispatcher class

According to the documentation, the addEventListener[17] method that is defined in the **EventDispatcher** class " *Registers an event listener object with an* ***EventDispatcher*** *object so that the listener receives notification of an event."*

Parameters of the addEventListener method

This method requires the following five parameters, the last three of which have default values:

1. type:String
2. listener:Function
3. useCapture:Boolean = false
4. priority:int = 0
5. useWeakReference:Boolean = false

Will concentrate on the first two parameters

I will concentrate on the first two parameters in this lesson. In addition, I will refer you to Understanding the AS3 Event Flow[18] by Jody Hall for an excellent discussion of the purpose and use of the third parameter.

The type of the event

The first parameter (p. 4) , which is the type of the event, usually looks something like the following in ActionScript syntax:

Event.ACTIVATE

The event-handler function

The second parameter (p. 4) is the name of a function or method that processes the event. This method must accept an **Event** object as its only parameter and must return void. A sample ActionScript signature for a suitable event handler method follows:

private function activateHandler(event:Event):void

Note that the actual type of the parameter may be any subclass of **Event** , such as **MouseEvent** for example. Note also that only the name of the function *(without parentheses)* is passed to the **addEventListener** method.

[11] http://livedocs.adobe.com/flex/3/langref/mx/controls/Button.html
[12] http://livedocs.adobe.com/flex/3/langref/flash/display/DisplayObject.html
[13] http://livedocs.adobe.com/flex/3/langref/flash/net/URLLoader.html
[14] http://livedocs.adobe.com/flex/3/langref/flash/display/DisplayObject.html
[15] http://livedocs.adobe.com/flex/3/langref/all-index-A.html
[16] http://livedocs.adobe.com/flex/3/langref/flash/events/EventDispatcher.html
[17] http://livedocs.adobe.com/flex/3/langref/flash/events/EventDispatcher.html#addEventListener()
[18] http://www.actionscript.org/resources/articles/860/1/Understanding-the-AS3-Event-Flow/Page1.html

Available to all subclasses of EventDispatcher

The **addEventListener** method is defined in the **EventDispatcher** class and inherited by all subclasses of that class. Therefore, the method can be called on any object instantiated from any subclass of the **EventDispatcher** class.

There are many subclasses of the EventDispatcher class

The **EventDispatcher** class has approximately seventy-five immediate subclasses that ultimately fan out to include hundreds and possibly thousands of individual classes. *(For example, the **Button** class is a subclass of the **EventDispatcher** class about seven levels down the inheritance hierarchy.)*

Many combinations are nonsensical

This means that hundreds of different event types can be registered on hundreds of different object types. However, many of those combinations of event types and object types make absolutely no sense at all.

The SolidColor class

For example, one of the subclasses of the **EventDispatcher** class is named **SolidColor** . You can register a handler for a **MouseEvent.CLICK** event on a **SolidColor** object with no obvious ill effects. There is no compiler error and no runtime error.

However, the registration of the event handler on the object has no effect. As far as I know, it is not possible to cause a **SolidColor** object to dispatch a **CLICK** event. Therefore, the **click** event handler will never be executed.

A weakness in the event model

In my opinion, the ability for the programmer to register event types on objects that will simply ignore events of that type is a major weakness in the ActionScript event model. Other programming languages such as Java provide more help in avoiding such programming errors.

Guarding against nonsensical combinations

How can you determine which combinations of classes and event types make sense and which do not?

As usual, your best friend is the documentation. For example, the ActionScript 3.0 Reference for the Adobe Flash Platform [19] allows you to click on a class name in the leftmost frame and read about that class in the rightmost frame. For every class that supports events, there is a hyperlink at the top of the rightmost frame labeled **Events** . Clicking on that hyperlink will expose all of the event types supported by objects of the class being viewed.

SolidColor events

For example, objects of the **SolidColor** class support only two types of events:

- flash.events.Event.ACTIVATE [20]
- flash.events.Event.DEACTIVATE [21]

Although you may be able to register event handlers for other types of events on an object of the **SolidColor** class, it doesn't make any sense to do so. The documentation tells us that there is no point in registering a **MouseEvent.CLICK** event on a **SolidColor** object.

The ACTIVATE and DEACTIVATE event types

Both of these event types are defined in the **EventDispatcher** class and are inherited by all classes whose objects have the ability to dispatch events *(other than subclasses of the **Proxy** class that I am ignoring in this lesson)* . They are both dispatched by the system as the result of certain runtime conditions that may be beyond the direct control of the user. I will explain a sample program later that uses these two event types.

The Flex Builder IDE is also helpful

Other useful tools for avoiding nonsensical combinations of classes and event types are the Flex Builder 3 and Flash Builder 4 IDEs. The IDEs provides popup hints at various points as you type ActionScript code. In some cases, the popup hints will list the types of events supported by the object on which you are registering an event listener.

[19]http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html
[20]http://livedocs.adobe.com/flex/3/langref/flash/events/EventDispatcher.html#event:activate
[21]http://livedocs.adobe.com/flex/3/langref/flash/events/EventDispatcher.html#event:deactivate

The DisplayObject class

Of the large number of immediate subclasses of the EventDispatcher [22] class, the one that will probably garner most of your attention in your early ActionScript programming experience is the class named DisplayObject [23] . According to the documentation:

> *"The DisplayObject class is the base class for all objects that can be placed on the display list. The display list manages all objects displayed in Flash Player or Adobe AIR."*

Flex components

All of the objects with which the user can have direct interaction are instantiated from subclasses of this class. For example, I believe that all of the components that appear in the **Components** tab in the **Design** view of Flex Builder 3 or Flash Builder 4 are subclasses of the **DisplayObject** class.

I also believe that all of those classes are grouped into packages such as the following:

- **mx.controls**
- **mx.containers**
- **mx.modules**
- **mx.charts**

There are about sixty-five classes in the **Components** tab of Flex Builder 3, and those are the classes that usually involve direct user interaction. That leaves many more classes that support events that don't usually involve direct user interaction.

Events that don't involve direct user interaction

Even some of the classes that are subclasses of **DisplayObject** support events that don't involve direct user interaction such as the **activate** and **deactivate** events.

Events that don't involve user interaction are usually events that are dispatched because of some change of state within the program. For example, it is possible to register event listeners on object properties and cause other objects to be notified when the value of such properties change.

2.4 Preview

I will present and explain two programs in the remainder of this lesson. The first program named **ActivateEvent01** provides a relatively simple illustration of servicing events that are dispatched independently of direct user interaction.

The second program named **Effects03** is somewhat more substantial. It illustrates the servicing of events that are dispatched as a result of direct user interaction as well as events that are dispatched independently of direct user interaction.

3 Discussion and sample code

A simple MXML file

Both of the programs that I will explain in this lesson use the same simple MXML file shown in Listing 1 and also in Listing 14.

Listing 1: Common code for the MXML file.

```
    <?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
```

[22] http://livedocs.adobe.com/flex/3/langref/flash/events/EventDispatcher.html
[23] http://livedocs.adobe.com/flex/3/langref/flash/display/DisplayObject.html

```
  <cc:Driver/>
</mx:Application>
```

This MXML code simply instantiates a new object of the **Driver** class in the **cc** namespace. Beyond that point, all program behavior is controlled by ActionScript code.

3.1 The program named ActivateEvent01

This is probably the most fundamental of all event-driven ActionScript programs. This program illustrates the **activate** and **deactivate** events. According to the EventDispatcher [24] documentation, the **activate** event is *"Dispatched when the Flash Player or AIR application gains operating system focus and becomes active."*

Similarly, the **deactivate** event is *"Dispatched when the Flash Player or AIR application loses the operating system focus and is becoming inactive."*

Program screen graphics

The screen output for this program is shown in Figure 1. This output doesn't change during the running of the program.

[24] http://livedocs.adobe.com/flex/3/langref/flash/events/EventDispatcher.html

Program ActivateEvent01 screen graphics.

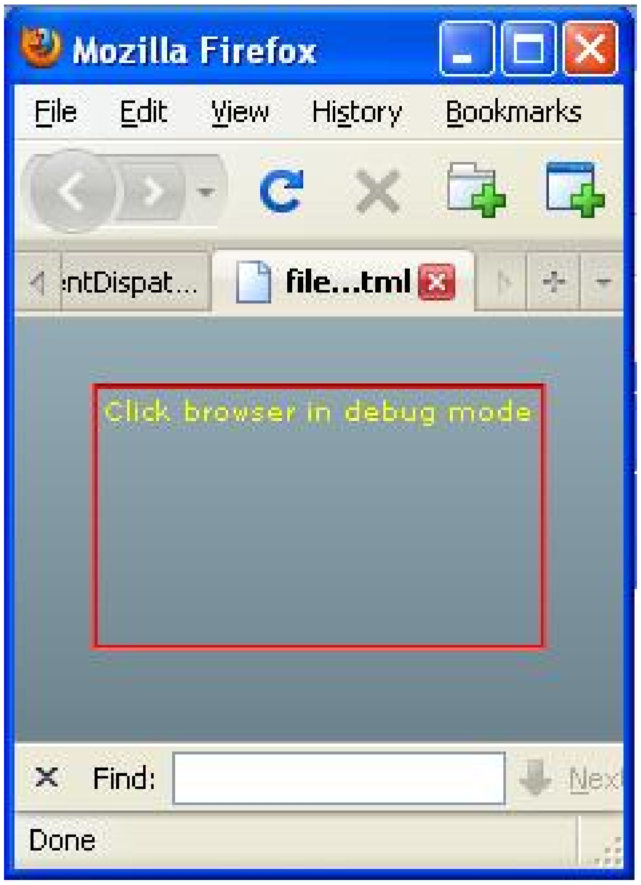


Figure 1: Program ActivateEvent01 screen graphics.

Must run in debug mode

This program uses calls to the **trace** function to produce output text on the console screen. Therefore, you will need to compile and run the program in debug mode in the IDE to get matching results.

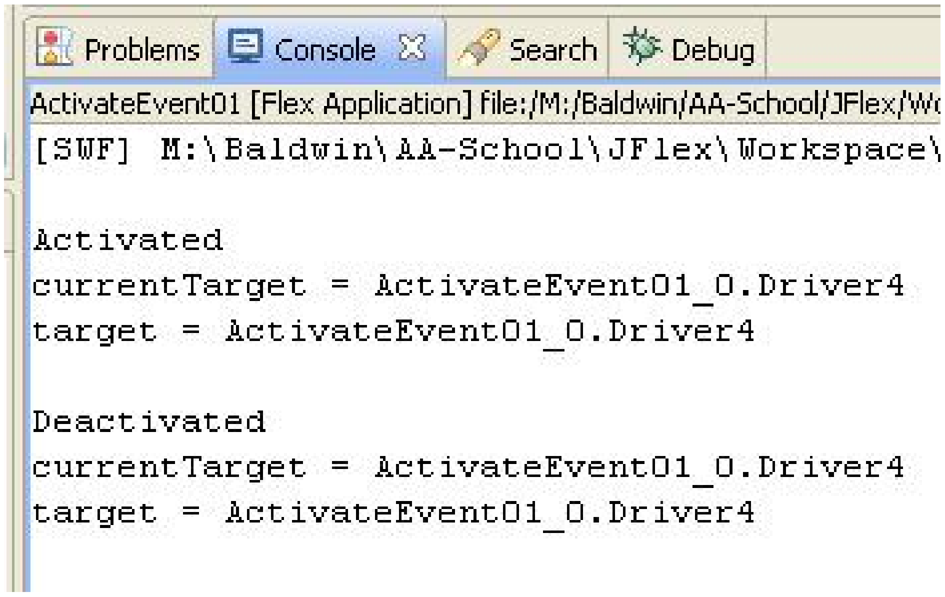
Gain and then lose operating system focus

If you start the program in debug mode, click somewhere inside the Flash window and then click somewhere in another window or on the desktop, you will cause the Flash Player to first gain and then lose the operating system focus. In other words, on the first click inside the Flash window, the Flash Player will become the active program. On the second click in another window, another program will become the active program.

Debug output in the system console

When you do that, output similar to that shown in Figure 2 should appear in the IDE console.

Debug output in the system console.



```
ActivateEvent01 [Flex Application] file:/M:/Baldwin/AA-School/JFlex/Wo
[SWF] M:\Baldwin\AA-School\JFlex\Workspace\

Activated
currentTarget = ActivateEvent01_0.Driver4
target = ActivateEvent01_0.Driver4

Deactivated
currentTarget = ActivateEvent01_0.Driver4
target = ActivateEvent01_0.Driver4
```

Figure 2: Debug output in the system console.

A snapshot of the Flex Builder 3 console

Figure 2 shows a snapshot of the console window in the Flex Builder 3 IDE when the above procedure was executed in debug mode.

The three lines of text beginning with the word *Activated* were produced when the Flash Player gained operating system focus. The three lines of text beginning with the word *Deactivated* were produced when the Flash Player lost operating system focus.

The currentTarget and target properties of the Event object

For an explanation of the **currentTarget** and the **target** properties shown in Figure 2, see the excellent article titled Understanding the AS3 Event Flow [25] by Jody Hall. As you will see in the next sample program, this information can be used to identify the component that dispatched the event.

Will discuss in fragments

I will explain the code for **ActivateEvent01** in fragments. A complete listing of the class file is provided in Listing 15 near the end of the lesson.

Beginning of Driver class for ActivateEvent01

Listing 2 shows the beginning of the **Driver** class for **ActivateEvent01** .

Listing 2: Beginning of Driver class for ActivateEvent01.

```
package CustomClasses{
  import flash.events.Event;
  import mx.containers.VBox;
```

[25] http://www.actionscript.org/resources/articles/860/1/Understanding-the-AS3-Event-Flow/Page1.html

```
import mx.controls.Label;

public class Driver extends VBox{
  public function Driver(){
    setStyle("borderStyle","inset");
    setStyle("borderColor",0xFF0000);
    height=100;

    var label:Label = new Label();
    label.text = "Click browser in debug mode"
    label.setStyle("color",0xFFFF00);
    addChild(label);

    addEventListener(Event.ACTIVATE,activateHandler);

    addEventListener(Event.DEACTIVATE,
                                deActivateHandler);
  }//end constructor
```

Extends the VBox class

The entire program is written in the **Driver** class and illustrates the **activate** and **deactivate** events of the **EventDispatcher** class.

This class extends the **VBox** class, so an object of this class is a **VBox** object.

Register event listeners

The first statement beginning with **addEventListener** registers an event listener on the **VBox** object causing it to execute the method named **activateHandler** when the Flash Player gains operating system focus and becomes the active program.

The second statement beginning with **addEventListener** registers an event listener on the **VBox** causing it to execute the **deActivateHandler** method when the Flash Player loses operating system focus and is no longer the active program.

The event handler methods

The two event handler methods are shown in Listing 3.

Listing 3: The event handler methods.

```
  private function activateHandler(event:Event):void{
      trace("\nActivated\ncurrentTarget = "
            + event.currentTarget
            + "\ntarget = "
            + event.target);
  }//end activateHandler
  //-----------------------------=-------------------//

  private function deActivateHandler(event:Event):void{
      trace("\nDeactivated\ncurrentTarget = "
            + event.currentTarget
            + "\ntarget = "
            + event.target);
  }//end deActivateHandler
  }//end class
}//end package
```

Output produced by the trace function

When the methods are executed in debug mode, the call to the **trace** function extracts and displays property values from the incoming **Event** object.

This causes the material shown in Figure 2 to be displayed in the system console window each time the Flash Player gains and then loses the operating system focus.

Listing 3 also signals the end of the class and the end of the program named **ActivateEvent01** .

3.2 The program named Effects03

This is a much more substantial program, which is based on a program in the Flex documentation [26] . That program is written mostly in Flex MXML and contains the minimum amount of ActionScript code necessary to provide the desired behavior.

Online versions of the two programs

An online executable version of the program is available following the source code listing at the above site [27] . I encourage you to run it and observe its behavior.

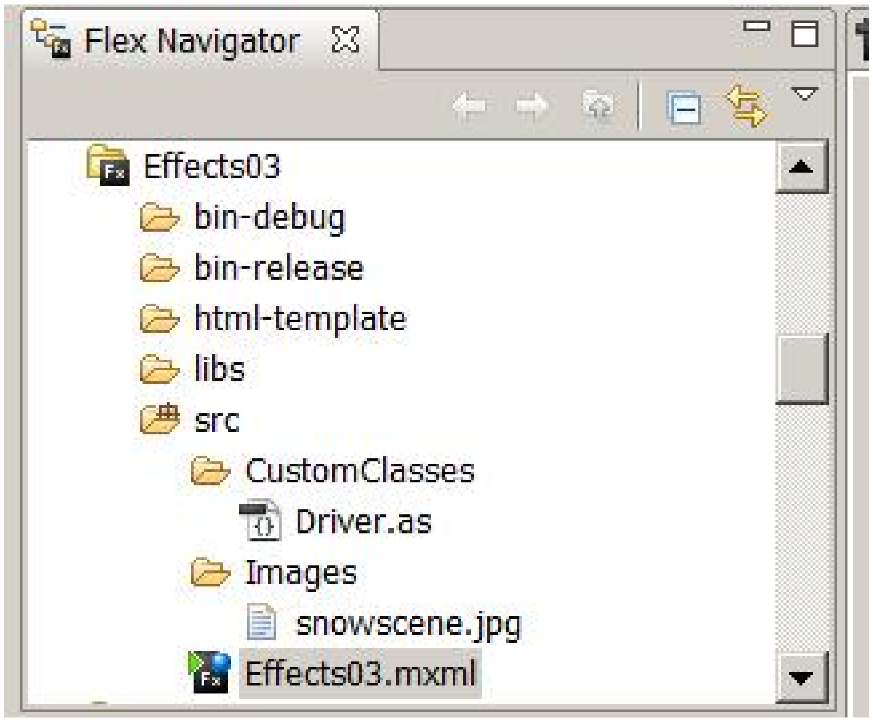
This version of the program is written almost entirely in ActionScript and contains only enough Flex MXML to make it possible to launch the program from within an HTML document. I also encourage you to run [28] this version and observe its behavior.

Project file structure for Effects03

The project file structure for the program is shown in Figure 3, which shows a snapshot of the Flex Builder 3 Navigator panel.

[26]http://livedocs.adobe.com/flex/3/langref/mx/effects/Effect.html#includeExamplesSummary
[27]http://livedocs.adobe.com/flex/3/langref/mx/effects/Effect.html#includeExamplesSummary
[28]http://cnx.org/content/m34452/latest/Effects03.html

Project file structure for Effects03.



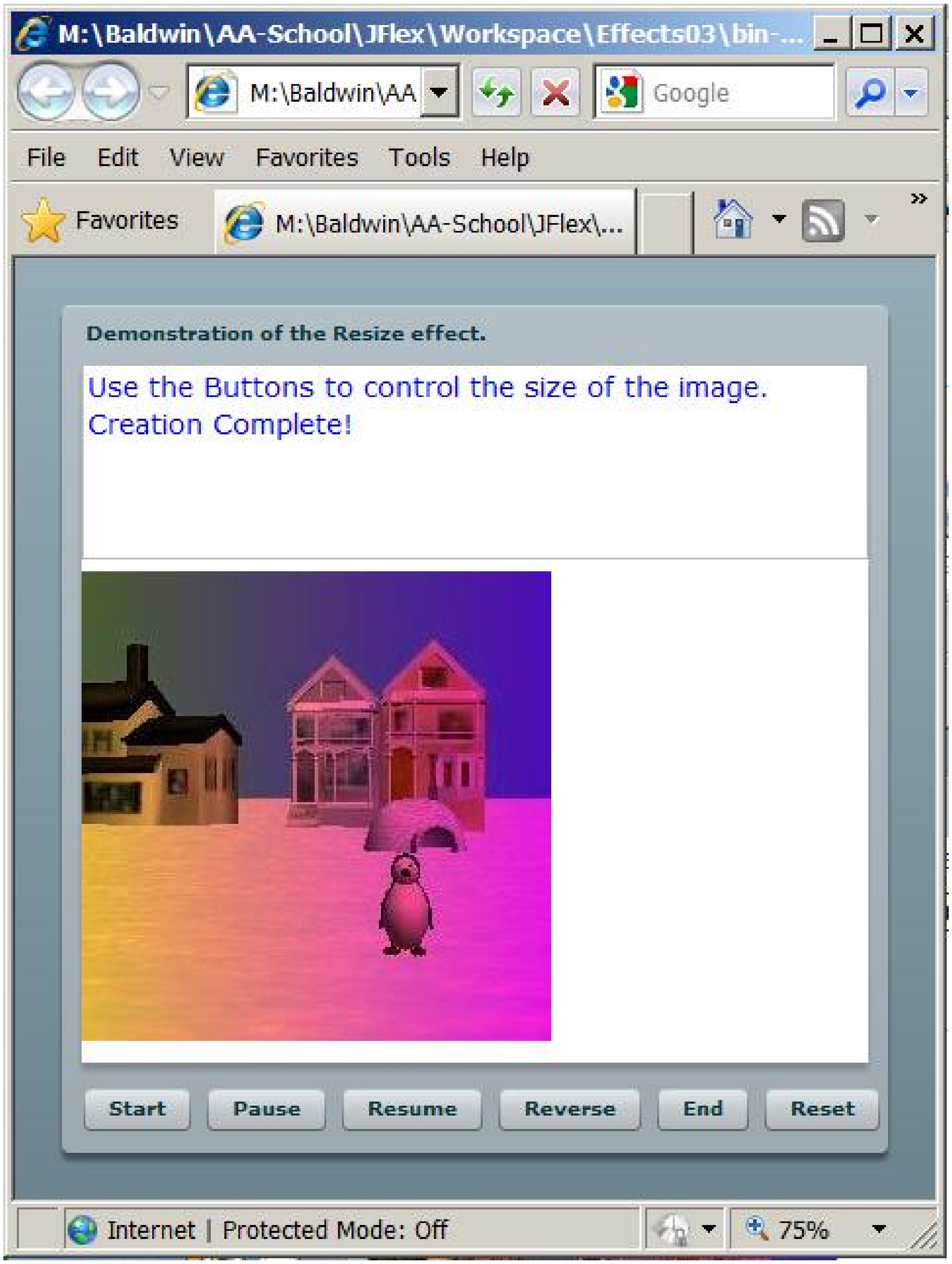
```
Effects03
    bin-debug
    bin-release
    html-template
    libs
    src
        CustomClasses
            Driver.as
        Images
            snowscene.jpg
        Effects03.mxml
```

Figure 3: Project file structure for Effects03.

Screen display at startup

The screen display at startup is shown in Figure 4.

Screen display at startup.


M:\Baldwin\AA-School\JFlex\Workspace\Effects03\bin-...
M:\Baldwin\AA
Google
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JFlex\...
Start
Pause
Resume
Reverse
End
Reset
Internet | Protected Mode: Off
75%

Demonstration of the Resize effect.

Use the Buttons to control the size of the image.
Creation Complete!

Figure 4: Screen display at startup.

A creationComplete event

By the time that the snapshot shown in Figure 4 was taken, the application had already dispatched a **creationComplete** event and the event handler registered to listen for that event had caused the text "Creation Complete!" to be displayed in a **TextArea** object near the top of the browser window.

What is a creationComplete event?

According to the documentation [29] , this event is dispatched *"when the component has finished its construction, property processing, measuring, layout, and drawing."* Therefore, other than the fact that the user starts the program running, the dispatching of this event type is completely beyond the control of the user.

Behavior of the program

This program uses a Resize [30] object from the mx.effects [31] package to cause the size of the image of the penguin shown in Figure 4 to be reduced to one-fourth of its original size over a period of 10 seconds when the user clicks the **Start** button shown at the bottom of Figure 4.

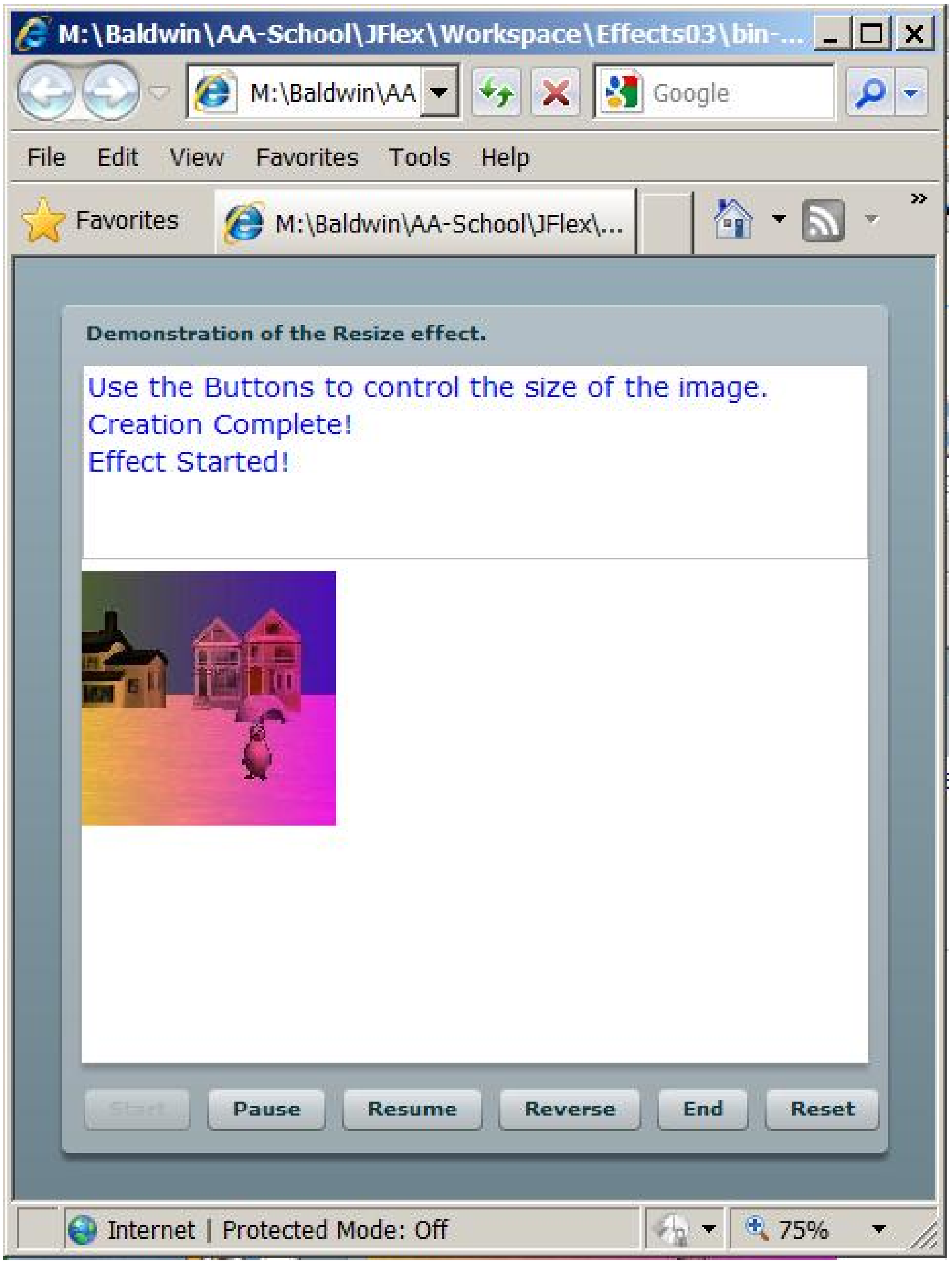
The behavior of the effect resulting from clicks on the other buttons at the bottom of Figure 4 generally matches the labels on the buttons.

Screen display after clicking Start and Pause

Figure 5 shows the screen display after clicking the **Start** button, allowing the effect to run for a few seconds, and then clicking the **Pause** button.

[29] http://livedocs.adobe.com/flex/3/langref/mx/core/UIComponent.html#event:creationComplete
[30] http://livedocs.adobe.com/flex/3/langref/mx/effects/Resize.html
[31] http://livedocs.adobe.com/flex/3/langref/mx/effects/package-detail.html

Screen display after clicking Start and Pause.


M:\Baldwin\AA-School\JFlex\Workspace\Effects03\bin-...
M:\Baldwin\AA
Google
File
Edit
View
Favorites
Tools
Help
Favorites
M:\Baldwin\AA-School\JFlex\...
Start
Pause
Resume
Reverse
End
Reset
Internet | Protected Mode: Off
75%

Demonstration of the Resize effect.

Use the Buttons to control the size of the image.
Creation Complete!
Effect Started!

Figure 5: Screen display after clicking Start and Pause.

Servicing an effectStart event

As you will see later, clicking the **Start** button causes a method named **play** to be called on an object of the **Resize** class.

The **Resize** object dispatches an **effectStart** event whenever it starts playing an effect. An event handler, registered on the **Resize** object, causes the text **"Effect Started!"** to be displayed in the text area near the top of Figure 5.

Doesn't depend on user interaction

Even though the user clicked the **Start** button to cause the **play** method to be called on the **Resize** object in this case, the dispatching of the **effectStart** event is generally independent of user interaction and is triggered by a change of state within the program.

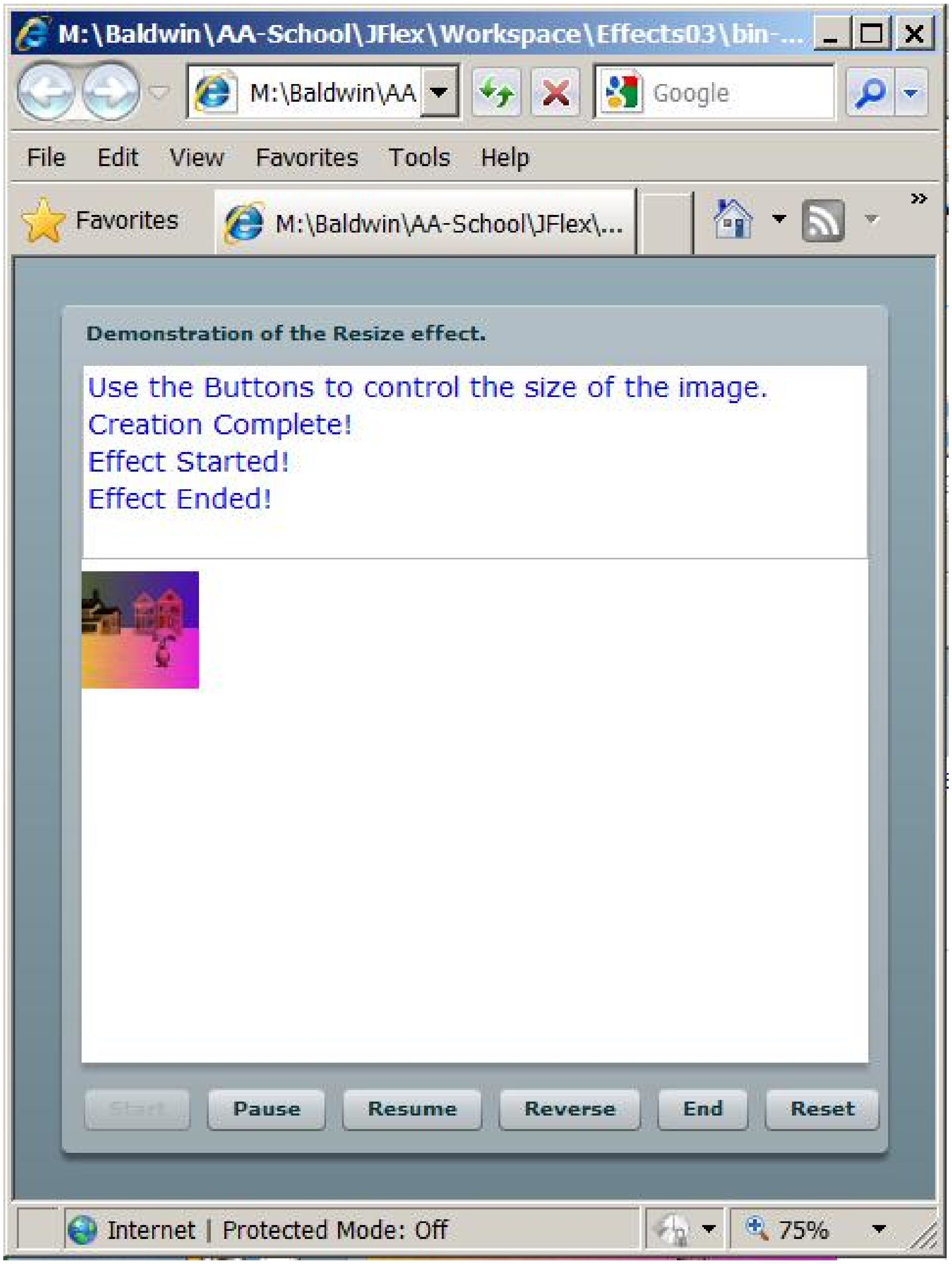
Could play the effect based on the time of day

For example, the **play** method could just as well have been called on the **Resize** object by another event handler that has determined that the time on the system clock has just struck midnight. In that case, the user could be home fast asleep and the **effectStart** event would still be dispatched.

Screen display after the effect has run to completion

The screen display in Figure 6 was captured after the user clicked the **Resume** button and allowed the effect to run to completion.

Screen display after the effect has run to completion.


M:\Baldwin\AA-School\JFlex\Workspace\Effects03\bin-...
M:\Baldwin\AA
Google
File
Edit
View
Favorites
Tools
Help
Favorites
M:\Baldwin\AA-School\JFlex\...
Demonstration of the Resize effect.
Start
Pause
Resume
Reverse
End
Reset
Internet | Protected Mode: Off
75%

Use the Buttons to control the size of the image.
Creation Complete!
Effect Started!
Effect Ended!

Figure 6: Screen display after the effect has run to completion.

Servicing an effectEnd event

The **Resize** object dispatches an **effectEnd** event when it finishes playing an effect. An event handler, registered on the **Resize** object, caused the text **"Effect Ended!"** to be displayed in the text area near the top of Figure 5.

Doesn't depend on user interaction

As before, the dispatching of the **effectEnd** event is generally independent of user interaction. Instead, it is triggered by a change of state within the program. In this case, at least ten seconds has elapsed since the user clicked the **Start** button to cause the effect to be played. Once again, the **play** method could just as well have been called on the **Resize** object by another event handler that has determined that the time on the system clock has struck midnight.

The MXML file

The MXML file used to launch this program as a Flex application is shown in Listing 14. The MXML code in Listing 14 simply instantiates an object of the **Driver** class file shown in the **CustomClasses** folder in Figure 2. From that point forward, the behavior of the program is entirely controlled by the ActionScript code that begins in Listing 4.

Event-driven aspects

This program may be interesting to you from two different perspectives. First, the event-driven aspects of the program illustrate the earlier discussion of the different types of events.

Some of the events that are serviced by the program, *(such as **click** events on the buttons)* , are dispatched as the direct result of user interaction. The remainder of the events, such as a **creationComplete** event, are dispatched independently of direct user interaction.

Correspondence between ActionScript and MXML code

The second interesting aspect of the program is that it illustrates that many things can be done using either Flex MXML code or ActionScript code.

The overall structure of this program is not identical to the structure of the program shown in the documentation [32] . However, it is similar enough that you should be able to map the MXML code from the program in the documentation to the ActionScript code in this program and understand how the two relate to one another.

Will explain in fragments

I will break this program down and explain it in fragments. Aside from the MXML file shown in Listing 14, the program consists of a single class named **Driver** . A complete listing of the **Driver** class file is provided in Listing 16 near the end of the lesson.

Beginning of the class named Driver

The Driver class begins in Listing 4. Note that this class extends the **Panel** class, causing this program to have the same overall appearance as the program provided in the documentation [33] .

Listing 4: Beginning of Driver class for Effects03.

```
 package CustomClasses{
import flash.events.MouseEvent;
import mx.containers.ControlBar;
import mx.containers.Panel;
import mx.controls.Button;
import mx.controls.Image;
import mx.controls.TextArea;
import mx.effects.Resize;
import mx.events.EffectEvent;
import mx.events.FlexEvent;
```

[32]http://livedocs.adobe.com/flex/3/langref/mx/effects/Effect.html#includeExamplesSummary
[33]http://livedocs.adobe.com/flex/3/langref/mx/effects/Effect.html#includeExamplesSummary

```
  public class Driver extends Panel{
    //Instantiate and save references to all of the
    // objects required by the program.
    private var resize:Resize = new Resize();
    private var textOut:TextArea = new TextArea();
    private var image:Image = new Image();
    private var bar:ControlBar = new ControlBar();
    private var startButton:Button = new Button();
    private var pauseButton:Button = new Button();
    private var resumeButton:Button = new Button();
    private var reverseButton:Button = new Button();
    private var endButton:Button = new Button();
    private var resetButton:Button = new Button();
```

In addition to containing all of the required import directives, the code in Listing 4 instantiates and saves references to all of the objects required by the program.

Beginning of the constructor for Effects03

The constructor begins in Listing 5.

Listing 5: Beginning of the constructor for Effects03.

```
    public function Driver(){//constructor
  this.title="Demonstration of the Resize effect.";
  this.percentWidth = 100;
  this.percentHeight = 100;

  this.addEventListener(FlexEvent.CREATION_COMPLETE,
                        creationCompleteHandler);
```

The code in Listing 5 sets several properties on the **Panel** object, which is the object with the light gray background shown in Figure 4. The **Panel** object is displayed in the browser window, which has a slightly darker gray background color in Figure 4.

Register an event listener

The last statement in Listing 5 registers an event listener named **creationCompleteHandler** to service creationComplete [34] events dispatched by the **Panel** object.

As you will see later, the event handler causes the text **"Creation Complete!"** to be displayed in a **TextArea** object near the top of the **Panel** in Figure 4 *when the* ***Panel*** *object and all of its children have been constructed, initialized, and drawn.*

Prepare the TextArea object and add it to the Panel

Listing 6 sets some properties and some styles on the **TextArea** object and adds it near the top of the **Panel** as shown in Figure 4.

Listing 6: Prepare the TextArea object and add it to the Panel.

```
      textOut.percentWidth = 100;//percent
  textOut.height = 100;//pixels
  textOut.setStyle("color","0x0000FF");
  textOut.setStyle("fontSize",14);
  textOut.text = "Use the Buttons to control "
```

[34] http://livedocs.adobe.com/flex/3/langref/mx/core/UIComponent.html#event:creationComplete

```
                        + "the size of the image.";
addChild(textOut);
```

This object will be used to display messages that track the progress of the program as the user clicks the buttons at the bottom of Figure 4. If the text area becomes full, a vertical scroll bar will automatically appear on the right side of the text area.

Prepare an embedded image and add it to the Panel

Listing 7 prepares an embedded image, loads it into an Image object, and adds the Image object to the **Panel** .

Listing 7: Prepare an embedded image and add it to the Panel.

```
    [Embed("/Images/snowscene.jpg")]
var img:Class;
image.load(img);

addChild(image);
```

Comparable MXML code

In case you're interested, the ActionScript code in Listing 7 is essentially comparable to the MXML code in Figure 7.

Comparable MXML code.

```
   <mx:Image id="img"
source="@Embed(
source='/Images/snowscene.jpg')"/>
```

Figure 7: Comparable MXML code.

Note, however, that in order to cause Figure 7 to fit into this narrow publication format, I inserted an extra line break character ahead of the second occurrence of the word "source." An MXML parser may not be willing to accept this line break.

Add the button bar to the panel

The six buttons at the bottom of Figure 4 are contained in an object of the class ControlBar [35] , which was instantiated in Listing 4.

Listing 8 adds the **ControlBar** container to the **Panel** object.

Listing 8: Add the button bar to the panel.

```
    addChild(bar);
```

Prepare the six buttons for use

The code in Listing 9:

- Adds labels to each of the six buttons at the bottom of Figure 4.

[35]http://livedocs.adobe.com/flex/3/langref/mx/containers/ControlBar.html

- Registers the same event handler to service **click** events on each of the six buttons.
- Adds each of the six buttons to the button bar shown at the bottom of Figure 4.

Listing 9: Prepare the six buttons for use.

```
    //Set text on the six buttons.
startButton.label = "Start";
pauseButton.label = "Pause";
resumeButton.label = "Resume";
reverseButton.label = "Reverse";
endButton.label = "End";
resetButton.label = "Reset";

//Register a click listener on each button
startButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
pauseButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
resumeButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
reverseButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
endButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
resetButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);

//Add the six buttons to the button bar
bar.addChild(startButton);
bar.addChild(pauseButton);
bar.addChild(resumeButton);
bar.addChild(reverseButton);
bar.addChild(endButton);
bar.addChild(resetButton);
```

Configure the Resize object

The resize effect that I described earlier (p. 14) is accomplished by calling the **play** method on the object of the class **Resize** that was instantiated in Listing 4.

Listing 10 targets that effect to the image shown in Figure 4. Listing 10 also specifies the final size of the image and the ten-second duration during which the resize effect will play. *(Note that the original size of the image is 240x240 pixels and it will be resized to one-fourth of its original size.)*

Listing 10: Configure the Resize object.

```
resize.target = image;
resize.widthTo = 60;
resize.heightTo = 60;
resize.duration = 10000;

resize.addEventListener(
               EffectEvent.EFFECT_END,endEffectHandler);
      resize.addEventListener(
            EffectEvent.EFFECT_START,startEffectHandler);

    } //end constructor
```

Register two event listeners

The first four statements in Listing 10 would be sufficient to play the effect if that was all that I wanted to do. In addition, however, my objective is to illustrate the servicing of events that are dispatched due to changes of state within the program.

Seven different types of events

A **Resize** object dispatches seven different types of events including the following two:

- effectStart [36] - Dispatched when the effect starts playing.
- effectEnd [37] - Dispatched when the effect finishes playing, either when the effect finishes playing or when the effect has been interrupted by a call to the **end()** method.

The last two statements in Listing 10 register event listeners on the **Resize** object for both of those types of events.

Listing 10 also signals the end of the constructor for the **Driver** class.

Event handlers registered on the Resize object

The two event handlers registered on the **Resize** object by Listing 10 are shown in Listing 11.

Listing 11: Event handlers registered on the Resize object.

```
    private function startEffectHandler(
                                 event:EffectEvent):void{
        textOut.text += "\nEffect Started!";
    } //end event handler
    //----------------------------------------------------//


    private function endEffectHandler(
                                 event:EffectEvent):void{
        textOut.text += "\nEffect Ended!";
    } //end event handler
```

The first method named **startEffectHandler** is executed each time the **Resize** object dispatches an **effectStart** event. The second method named **endEffectHandler** is executed each time the **Resize** object dispatches an **effectEnd** event.

Output text at the start and the end of the effect

The text in the text area at the top of Figure 5 shows the result of executing the **startEffectHandler** method from Listing 11.

The text in the text area at the top of Figure 6 shows the result of executing the **endEffectHandler** method from Listing 11.

(Note the small size of the image at the end of the resize effect in Figure 6 as compared to the size of the image at the beginning of the resize effect in Figure 4.)

[36] http://livedocs.adobe.com/flex/3/langref/mx/effects/Effect.html#event:effectStart
[37] http://livedocs.adobe.com/flex/3/langref/mx/effects/Effect.html#event:effectEnd

Methods of the Resize class

The **Resize** class defines several methods including the following six *(in alphabetical order)* :

- **end** - Interrupts an effect that is currently playing, and jumps immediately to the end of the effect.
- **pause** - Pauses the effect until you call the **resume** method.
- **play** - Begins playing the effect.
- **resume** - Resumes the effect after it has been paused by a call to the **pause** method.
- **reverse** - Plays the effect in reverse *(if the effect is currently playing)* , starting from the current position of the effect.
- **stop** - Stops the effect, leaving the effect targets in their current state.

Common event handler for the buttons

Listing 12 shows a common event handler that is used to service **click** events on all six of the buttons at the bottom of Figure 4.

Listing 12: Common event handler for the buttons.

```
private function btnHandler(event:MouseEvent):void{
  if (event.target == startButton) {
    resize.play();//start the effect
    startButton.enabled = false;
  }else if(event.target == pauseButton){
    resize.pause();//pause the effect
  }else if(event.target == resumeButton){
    resize.resume();//resume the effect after a pause
  }else if(event.target == reverseButton){
    resize.reverse();//reverse the effect
  }else if(event.target == endButton){
    resize.end();//end the effect prematurely
  }else{//reset the program to starting conditions
    resize.end();
    image.width=240;
    image.height=240;
    startButton.enabled=true;
  } //end else

} //end btnHandler
```

Identify the button that dispatched the event

Listing 12 extracts and uses the **target** property of the incoming **MouseEvent** object to identify which of the six buttons dispatched the **click** event that caused the method to be executed.

Call a corresponding method on the Resize object

For each of the first five buttons shown in Figure 4, Listing 12 calls a corresponding method from the above list (p. 23) on the **Resize** object. *(Note that clicking the **Start** button causes the **Start** button to be disabled by the code in Listing 12. Note also that none of the buttons call the **stop** method from the above list.)*

Service the Reset button

The sixth button, labeled **Reset** also calls the **end** method from the above list to cause the effect to immediately jump to the end. Then it executes some additional code to restore the image to its original size and to re-enable the **Start** button.

Service the creationComplete event dispatched by the Panel

That brings us to the final event handler method and the end of the program. The method shown in Listing 13 is executed when the **Panel** dispatches a **creationComplete** event.

Listing 13: Event handler for the creationComplete event dispatched by the Panel.

```
    private function creationCompleteHandler(
                              event:FlexEvent):void{
    textOut.text += "\nCreation Complete!";
} //end event handler
```

As you saw in Figure 4, the code in this method causes the text "Creation Complete!" to be displayed in the text area when the **Panel** and all of its children have been created, initialized, and drawn.

4 Run the program

I encourage you to run [38] this program from the web. Then copy the code from Listing 14 through Listing 16. Use that code to create Flex projects. Compile and run the projects. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.1 Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

5 Complete program listings

Complete listings of the programs discussed in this lesson are provided in Listing 14 through Listing 16.

Listing 14: The common MXML code.

```
    <?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
  <cc:Driver/>
</mx:Application>
```

Listing 15: Driver class for ActivateEvent01.

```
    //Illustrates the activate and deactivate events of the
// EventDispatcher class. Must run in debug mode to see
// the text output.
// Click on the browser to cause the Flash player
// to gain focus and fire an activate event.
// Click on the desktop to cause the Flash player to
// lose focus and fire a deactivate event.
```

[38] http://cnx.org/content/m34452/latest/Effects03.html

```
package CustomClasses{
  import flash.events.Event;
  import mx.containers.VBox;
  import mx.controls.Label;

  public class Driver extends VBox{
    public function Driver(){
      setStyle("borderStyle","inset");
      setStyle("borderColor",0xFF0000);
      height=100;

      var label:Label = new Label();
      label.text = "Click browser in debug mode"
      label.setStyle("color",0xFFFF00);
      addChild(label);

      addEventListener(Event.ACTIVATE,activateHandler);
      addEventListener(Event.DEACTIVATE,
                                      deActivateHandler);
    }//end constructor


    private function activateHandler(event:Event):void{
        trace("\nActivated\ncurrentTarget = "
              + event.currentTarget
              + "\ntarget = "
              + event.target);
    }//end activateHandler

    private function deActivateHandler(event:Event):void{
        trace("\nDeactivated\ncurrentTarget = "
              + event.currentTarget
              + "\ntarget = "
              + event.target);
    }//end deActivateHandler

  }//end class
}//end package
```

Listing 16: Driver class for Effects03.

```
    /*This is an ActionScript version of a program that is
similar to the sample MXML program in the documentation
for the Effect class at:
http://livedocs.adobe.com/flex/3/langref/mx/effects
/Effect.html. However, several changes were made to the
behavior of the program to make it more suitable for the
instructional purpose.
*/
```

```
package CustomClasses{
  import flash.events.MouseEvent;
  import mx.containers.ControlBar;
  import mx.containers.Panel;
  import mx.controls.Button;
  import mx.controls.Image;
  import mx.controls.TextArea;
  import mx.effects.Resize;
  import mx.events.EffectEvent;
  import mx.events.FlexEvent;


  public class Driver extends Panel{
    //Instantiate and save references to all of the
    // objects needed by the program.
    private var resize:Resize = new Resize();
    private var textOut:TextArea = new TextArea();
    private var image:Image = new Image();
    private var bar:ControlBar = new ControlBar();
    private var startButton:Button = new Button();
    private var pauseButton:Button = new Button();
    private var resumeButton:Button = new Button();
    private var reverseButton:Button = new Button();
    private var endButton:Button = new Button();
    private var resetButton:Button = new Button();
    //--------------------------------------------------//

    public function Driver(){//constructor
      this.title="Demonstration of the Resize effect.";
      this.percentWidth = 100;
      this.percentHeight = 100;
      this.addEventListener(FlexEvent.CREATION_COMPLETE,
                          creationCompleteHandler);

      //Set textOut properties and add to the panel.
      textOut.percentWidth = 100;//percent
      textOut.height = 100;//pixels
      textOut.setStyle("color","0x0000FF");
      textOut.setStyle("fontSize",14);
      textOut.text = "Use the Buttons to control "
                            + "the size of the image.";
      addChild(textOut);

      //Prepare an embedded image and add the Image
      // object to the panel.
      [Embed("/Images/snowscene.jpg")]
      var img:Class;
      image.load(img);
      addChild(image);
      //Add the button bar to the panel.
      addChild(bar);

      //Set text on the six buttons.
      startButton.label = "Start";
      pauseButton.label = "Pause";
      resumeButton.label = "Resume";
      reverseButton.label = "Reverse";
      endButton.label = "End";
      resetButton.label = "Reset";

      //Register a click listener on each button
      startButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
      pauseButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
      resumeButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
      reverseButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
      endButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);
      resetButton.addEventListener(
                      MouseEvent.CLICK,btnHandler);

      //Add the six buttons to the button bar
      bar.addChild(startButton);
      bar.addChild(pauseButton);
      bar.addChild(resumeButton);
      bar.addChild(reverseButton);
      bar.addChild(endButton);
      bar.addChild(resetButton);


      //Configure the Resize effect. Note that the
      // original size of the image is 240x240.
      resize.target = image;
      resize.widthTo = 60;
      resize.heightTo = 60;
      resize.duration = 10000;
      resize.addEventListener(
                EffectEvent.EFFECT_END,endEffectHandler);
      resize.addEventListener(
            EffectEvent.EFFECT_START,startEffectHandler);

    } //end constructor
    //--------------------------------------------------//

    //This common button handler is used to service click
    // event on all six of the buttons.
    private function btnHandler(event:MouseEvent):void{
      if (event.target == startButton) {
        resize.play();//start the effect
        startButton.enabled = false;
      }else if(event.target == pauseButton){
        resize.pause();//pause the effect
      }else if(event.target == resumeButton){
        resize.resume();//resume the effect after a pause
      }else if(event.target == reverseButton){
        resize.reverse();//reverse the effect
      }else if(event.target == endButton){
        resize.end();//end the effect prematurely
      }else{//reset the program to starting conditions
        resize.end();
        image.width=240;
        image.height=240;
        startButton.enabled=true;
      } //end else

    } //end btnHandler
    //--------------------------------------------------//

    //This event handler method is executed when the
    // effect ends.
    private function endEffectHandler(
                              event:EffectEvent):void{
        textOut.text += "\nEffect Ended!";
    } //end event handler
    //--------------------------------------------------//

    //This event handler method is executed when the
    // effect starts
    private function startEffectHandler(
                              event:EffectEvent):void{
        textOut.text += "\nEffect Started!";
    } //end event handler
    //--------------------------------------------------//

    //This event handler method is executed when the
    // application dispatches a creationComplete event.
    private function creationCompleteHandler(
                                event:FlexEvent):void{
        textOut.text += "\nCreation Complete!";
    } //end event handler
    //--------------------------------------------------//

  } //end class
} //end package
```

6 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Digging Deeper into ActionScript Events
- Files:
    - ActionScript0114\ActionScript0114.htm
    - ActionScript0114\Connexions\ActionScriptXhtml0114.htm

NOTE: **PDF disclaimer:** Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

-end-