The Flex Event Model

In Java, events and event handling occur by setting up special classes. In Flex, the entire framework is event-oriented making Flex a framework that supports loose-coupling among its components (a good thing!)

Using events

Using events in Flex is a two-step process. First, you write a function or class method, known as an event listener or event handler, that responds to events. The function often accesses the properties of the Event object or some other settings of the application state. The signature of this function usually includes an argument that specifies the event type being passed in.

The following example shows a simple event listener function that reports when a control triggers the event that it is listening for:

```xml
<?xml version="1.0"?><
<!-- events/SimpleEventHandler.mxml -->
    creationComplete="initApp();">  
    <mx:Script><![CDATA[
        import mx.controls.Alert;

        private function initApp():void {
            b1.addEventListener(MouseEvent.CLICK, myEventHandler);
        }

        private function myEventHandler(event:Event):void {
            Alert.show("An event occurred.");
        }
    ]]></mx:Script>

    <mx:Button id="b1" label="Click Me"/>

</mx:Application>
```

As you can see in this example, you also register that function or class method with a display list object by using the addEventListener() method.

Most Flex controls simplify listener registration by letting you specify the listener inside the MXML tag. For example, instead of using the addEventListener() method to specify a
listener function for the Button control’s click event, you specify it in the click attribute of the <mx:Button> tag:

```xml
<?xml version="1.0"?>
<!-- events/SimplerEventHandler.mxml -->
  <mx:Script><![CDATA[
    import mx.controls.Alert;
    
    private function myEventHandler(event:Event):void {
      Alert.show("An event occurred.");
    }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</mx:Application>
```

This is equivalent to the addEventListener() method in the previous code example. However, it is best practice to use the addEventListener() method. This method gives you greater control over the event by letting you configure the priority and capturing settings, and use event constants. In addition, if you use addEventListener() to add an event handler, you can use removeEventListener() to remove the handler when you no longer need it. If you add an event handler inline, you cannot call removeEventListener() on that handler.

Each time a control generates an event, Flex creates an Event object that contains information about that event, including the type of event and a reference to the dispatching control. To use the Event object, you specify it as a parameter in the event handler function, as the following example shows:

```xml
<?xml version="1.0"?>
<!-- events/EventTypeHandler.mxml -->
  <mx:Script><![CDATA[
    import mx.controls.Alert;
    
    private function myEventHandler(e:Event):void {
      Alert.show("An event of type "+e.type+" occurred.");
    }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</mx:Application>
```
If you want to access the Event object in an event handler that was triggered by an inline event, you must add the `event` keyword inside the MXML tag so that Flex explicitly passes it to the handler, as in the following:

```xml
<mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
```

You are not required to use the Event object in a handler function. The following example creates two event handler functions and registers them with the events of a ComboBox control. The first event handler, `openEvt()`, takes no arguments. The second event handler, `changeEvt()`, takes the Event object as an argument and uses this object to access the `value` and `selectedIndex` of the ComboBox control that triggered the event.

```xml
<?xml version="1.0"?>
<!-- events/MultipleEventHandlers.mxml -->
  <mx:Script><![CDATA[
    private function openEvt():void {
      forChange.text="";
    }

    private function changeEvt(e:Event):void {
      forChange.text =
        "Value: " + e.currentTarget.value + "\n" +
        "Index: " + e.currentTarget.selectedIndex;
    }
  ]]>]]></mx:Script>

  <mx:ComboBox open="openEvt()" change="changeEvt(event)">
    <mx:dataProvider>
      <mx:Array>
        <mx:String>AK</mx:String>
        <mx:String>AL</mx:String>
        <mx:String>AR</mx:String>
      </mx:Array>
    </mx:dataProvider>
  </mx:ComboBox>

  <mx:TextArea id="forChange" width="150" height="100"/>
</mx:Application>
```
Specifying the Event object

You specify the object in a listener function's signature as type Event, as the following example shows:

```actionscript
function myEventListener(e:Event):void { ... }
```

However, if you want to access properties that are specific to the type of event that was dispatched, you must instead specify a more specific event type, such as ToolTipEvent or KeyboardEvent, as the following example shows:

```actionscript
import mx.events.ToolTip
function myEventListener(e:ToolTipEvent):void { ... }
```

In some cases, you must import the event's class in your ActionScript block.

Most objects have specific events that are associated with them, and most of them can dispatch more than one type of event.

If you declare an event of type Event, you can cast it to a more specific type to access its event-specific properties. For more information, see Using event subclasses.

Accessing the currentTarget property

Event objects include a reference to the instance of the dispatching component (or target), which means that you can access all the properties and methods of that instance in an event listener. The following example accesses the id of the Button control that triggered the event:

```xml
<?xml version="1.0"?>
<!-- events/AccessingCurrentTarget.mxml -->
  <mx:Script><![CDATA[
    import mx.controls.Alert;

    private function myEventHandler(e:Event):void {
      Alert.show("The button "+ e.currentTarget.id + " was clicked.");
    }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</mx:Application>
```

You can access members of the currentTarget. If you do not cast the current target to a specific type, the compiler assumes that it is of type Object. Objects can have any property or
method because the Object type is dynamic in ActionScript. Therefore, when accessing methods and properties of the currentTarget, it is best practice to cast currentTarget to whatever class you anticipate will dispatch that event. This gives you strong type checking at compile time, and helps avoid the risk of throwing a run-time error.

The following example casts the current target to a TextInput class before calling the setSelection() method, but does not cast it before trying to set the tmesis property. The tmesis property does not exist on the TextInput class. This illustrates that you will get a run-time error but not a compile-time error when you try to access members that don't exist, unless you cast currentTarget to a specific type so that type checking can occur:

```xml
<?xml version="1.0"?>
<!-- events/InvokingOnCurrentTarget.mxml -->
    <mx:Script><![CDATA[
        import mx.core.UIComponent;
        private function tiHandler(e:Event):void {
            /* The following enforces type checking: */
            TextInput(e.currentTarget).setSelection(0,3);

            /* The following throws a run-time error but not a compile-time error: */
            e.currentTarget.tmesis = 4;
            */

            /*
            ... unless you cast it to the expected type like the following. Then
            the compiler throws an error.
            TextInput(e.currentTarget).tmesis = 4;
            */
        }
    ]]></mx:Script>

    <mx:TextInput id="ti1" click="tiHandler(event)"
        text="This is some text. When you click on this control, the first three characters are selected."/>

</mx:Application>
```

You could also cast currentTarget to UIComponent or some other more general class that still has methods of display objects. That way, if you don't know exactly which control will dispatch an event, at least you can ensure there is some type checking.

You can also access methods and properties of the target property, which contains a reference to the current node in the display list. For more information, see About the target and currentTarget properties.
Registering event handlers

There are several strategies that you can employ when you register event handlers with your Flex controls:

1. Define an event handler inline. This binds a call to the handler function to the control that triggers the event.
2. `<xml version="1.0">`
4. `<mx:Script><![CDATA[
5. import mx.controls.Alert;
6. private function myEventHandler(event:Event):void {
7.     Alert.show("An event occurred.");
8. }]]></mx:Script>`
9. `<mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>`
10. `</mx:Application>`
11. In this example, whenever the user clicks the Button control, Flex calls the myEventHandler() function.
12. For more information on defining event handlers inline, see Defining event listeners inline.

13. Use the addEventListener() method, as follows:
14. `<xml version="1.0">`
15. `<!-- events/SimpleEventHandler.mxml -->`
17. `<mx:Script><![CDATA[
18. import mx.controls.Alert;
19. private function initApp():void {
20.     b1.addEventListener(MouseEvent.CLICK, myEventHandler);
21. }]]></mx:Script>`
22. `<mx:Button id="b1" label="Click Me"/>`
23. `</mx:Application>`
As with the previous example, whenever the user clicks the Button control, Flex calls the myClickHandler() handler function. However, registering your event handlers using this method provides more flexibility. You can register multiple components with this event handler, add multiple handlers to a single component, or remove the handler. For more information, see Using the addEventListener() method.

Create an event handler class and register components to use the class for event handling. This approach to event handling promotes code reuse and lets you centralize event handling outside your MXML files. For more information on creating custom event handler classes, see Creating event handler classes.

Defining event listeners inline

The simplest method of defining event handlers in Flex applications is to point to a handler function in the component's MXML tag. To do this, you add any of the component's events as a tag attribute followed by an ActionScript statement or function call.

You add an event handler inline using the following syntax:

<mx:tag_name event_name="handler_function"/>

For example, to listen for a Button control's click event, you add a statement in the <mx:Button> tag's click attribute. If you add a function, you define that function in an ActionScript block. The following example defines the submitForm() function as the handler for the Button control's click event:

<mx:Script><![CDATA[
    function submitForm():void {
        // Do something.
    }
]]></mx:Script>

<mx:Button label="Submit" click="submitForm();"/>

Event handlers can include any valid ActionScript code, including code that calls global functions or sets a component property to the return value. The following example calls the trace() global function:

<mx:Button label="Get Ver" click="trace('The button was clicked');"/>

There is one special parameter that you can pass in an inline event handler definition: the event parameter. If you add the event keyword as a parameter, Flex passes the Event object and inside the handler function, you can then access all the properties of the Event object.
The following example passes the Event object to the `submitForm()` handler function and specifies it as type `MouseEvent`:

```xml
<?xml version="1.0"?>
    <mx:Script><![CDATA[
        import mx.controls.Alert;
        private function myEventHandler(event:MouseEvent):void {
            // Do something with the MouseEvent object.
            Alert.show("An event of type "+ event.type + ": occurred.");
        }
    ]]>></mx:Script>
    <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</mx:Application>
```

It is best practice to include the `event` keyword when you define all inline event listeners and to specify the most stringent Event object type in the resulting listener function (for example, specify `MouseEvent` instead of `Event`).

You can use the Event object to access a reference to the target object (the object that dispatched the event), the type of event (for example, `click`), or other relevant properties, such as the row number and value in a list-based control. You can also use the Event object to access methods and properties of the target component, or the component that dispatched the event.

Although you will most often pass the entire Event object to an event listener, you can just pass individual properties, as the following example shows:

```xml
<?xml version="1.0"?>
    <mx:Script><![CDATA[
        import mx.controls.Alert;
        private function myEventHandler(s:String):void {
            Alert.show("Current Target: "+ s);
        }
    ]]>></mx:Script>
    <mx:Button id="b1" label="Click Me"
        click="myEventHandler(event.currentTarget.id)"/>
</mx:Application>
```
Registering an event listener inline provides less flexibility than using the `addEventListener()` method to register event listeners. The drawbacks are that you cannot set the `useCapture` or `priority` properties on the Event object and that you cannot remove the listener once you add it.

**Using the `addEventListener()` method**

The `addEventListener()` method lets you register event listener functions with the specified control or object. The following example adds the `myClickListener()` function to the `b1` instance of a Button control. When the user clicks `b1`, Flex calls the `myClickListener()` method:

```javascript
b1.addEventListener(MouseEvent.CLICK, myClickListener);
```

The `addEventListener()` method has the following signature:

```javascript
componentInstance.addEventListener(
    event_type:String,
    event_listener:Function,
    use_capture:Boolean,
    priority:int,
    weakRef:Boolean
)
```

The `event_type` argument is the kind of event that this component dispatches. This can be either the event type String (for example, `click` or `mouseOut`) or the event type static constant (such as `MouseEvent.CLICK` or `MouseEvent.MOUSE_OUT`). This argument is required.

The constants provide an easy way to refer to specific event types. You should use these constants instead of the strings that they represent. If you misspell a constant name in your code, the compiler catches the mistake. If you instead use strings and make a typographical error, it can be harder to debug and could lead to unexpected behavior.

You should use the constants wherever possible. For example, when you are testing to see whether an Event object is of a certain type, use the following code:

```javascript
if (myEventObject.type == MouseEvent.CLICK) {/* your code here */}
```

Do not use the following code:

```javascript
if (myEventObject.type == "click") {/* your code here */}
```

The `event_listener` argument is the function that handles the event. This argument is required.
The `use_capture` parameter of the `addEventListener()` method lets you control the phase in the event flow in which your listener will be active. It sets the value of the `useCapture` property of the Event object. If `useCapture` is set to `true`, your listener is active during the capturing phase of the event flow. If `useCapture` is set to `false`, your listener is active during the targeting and bubbling phases of the event flow, but not during the capturing phase. The default value is determined by the type of event, but is `false` in most cases.

To listen for an event during all phases of the event flow, you must call `addEventListener()` twice, once with the `useCapture` parameter set to `true`, and again with `use_capture` set to `false`. This argument is optional. For more information, see Capturing phase.

The `priority` parameter sets the priority for that event listener. The higher the number, the sooner that event handler executes relative to other event listeners for the same event. Event listeners with the same priority are executed in the order that they were added. This parameter sets the `priority` property of the Event object. The default value is 0, but you can set it to negative or positive integer values. If several event listeners are added without priorities, the earlier a listener is added, the sooner it is executed. For more information on setting priorities, see Event priorities.

The `weakRef` parameter provides you with some control over memory resources for listeners. A strong reference (when `weakRef` is `false`) prevents the listener from being garbage collected. A weak reference (when `weakRef` is `true`) does not. The default value is `false`.

When you add a listener function and that function is invoked, Flex implicitly creates an Event object for you and passes it to the listener function. You must declare the Event object in the signature of your listener function.

If you add an event listener by using the `addEventListener()` method, you are required to declare an event object as a parameter of the `listener_function`, as the following example shows:

```actionscript
b1.addEventListener(MouseEvent.CLICK, performAction);
```

In the listener function, you declare the Event object as a parameter, as follows:

```actionscript
public function performAction(e:MouseEvent):void {
    ...
}
```

The following example defines a new handler function `myClickListener()`. It then registers the `click` event of the Button control with that handler. When the user clicks the button, Flex calls the `myClickHandler()` function.

```actionscript
b1.addEventListener(MouseEvent.CLICK, myClickListener);
```
<?xml version="1.0"?>
<!-- events/AddEventListenerExample.mxml -->
initialize="createListener()">
  <mx:Script><![CDATA[
import mx.controls.Alert;
private function createListener():void {
    b1.addEventListener(MouseEvent.CLICK, myClickHandler, false, 0);
}

private function myClickHandler(e:MouseEvent):void {
    Alert.show("The button was clicked.");
}
]]></mx:Script>
<mx:Button label="Click Me" id="b1"/>
</mx:Application>

Using addEventListener() inside an MXML tag

You can add event listeners with the addEventListener() method inline with the component definition. The following Button control definition adds the call to the addEventListener() method inline with the Button control's initialize property:

<?xml version="1.0"?>
<!-- events/CallingAddEventListenerInline.mxml -->
  <mx:Script><![CDATA[
import mx.controls.Alert;

private function myClickHandler(event:Event):void {
    Alert.show("The button was clicked.");
}
]]></mx:Script>

<mx:Button id='b1'
    label="Click Me"
    initialize='b1.addEventListener(MouseEvent.CLICK, myClickHandler, false, 1);' />
</mx:Application>

This is the equivalent of defining the event handler inline. However, defining a handler by using the addEventListener() method rather than setting click="handler_function" lets you set the value of the useCapture and priority properties of the Event object. Furthermore, you cannot remove a handler added inline, but when you use the addEventListener() method to add a handler, you can call the removeEventListener() method to remove that handler.
Using nested inner functions as event listeners

Rather than passing the name of an event listener function to the `addEventListener()` method, you can define an inner function (also known as a closure).

In the following example, the nested inner function is called when the button is clicked:

```xml
<!-- events/AddingInnerFunctionListener.mxml -->
.creationComplete="initApp()">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function initApp():void {
                b1.addEventListener("click", function(e:Event):void {
                    Alert.show("The button was clicked.");
                });
            }
        ]]>]
    </mx:Script>
    <mx:Button id='b1' label="Click Me"/>
</mx:Application>
```

Function closures are created any time a function is executed apart from an object or a class. They retain the scope in which they were defined. This creates interesting results when a function is passed as an argument or a return value into a different scope.

For example, the following code creates two functions: `foo()`, which returns a nested function named `rectArea()` that calculates the area of a rectangle, and `bar()`, which calls `foo()` and stores the returned function closure in a variable named `myProduct`. Even though the `bar()` function defines its own local variable `x` (with a value of 2), when the function closure `myProduct()` is called, it retains the variable `x` (with a value of 40) defined in function `foo()`. The `bar()` function therefore returns the product of the numbers in the TextInput controls, rather than 8.

```xml
<!-- events/FunctionReturnsFunction.mxml -->
.creationComplete="foo()">
    <mx:Script>
        <![CDATA[
            [Bindable]
            private var answer:String;
            private function foo():Function {
                var x:int = int(ti1.text);
                function rectArea(y:int):int { // function closure defined
                    return x * y;
                }
            }
        ]]>]
    </mx:Script>
    <mx:TextInput id='ti1' label="Enter a number" />
</mx:Application>
```
return rectArea;
}

private function bar():void {
var x:int = 2; // ignored
var y:int = 4; // ignored
var myProduct:Function = foo();
answer = myProduct(int(ti2.text)); // function closure called
}

</mx:Script>

<mx:Form width="107">
  <mx:FormItem label="X">
    <mx:TextInput id="ti1" text="10" width="37" textAlign="right"/>
  </mx:FormItem>
  <mx:FormItem label="Y" width="71">
    <mx:TextInput id="ti2" text="20" width="38" textAlign="right"/>
  </mx:FormItem>
  <mx:Label id="label1" text="{answer}" width="71" textAlign="right"/>
</mx:Form>

<mx:Button id='b1' label="Compute Product" click="bar()"/>
</mx:Application>

If the listener that you pass to addEventListener() method is a nested inner function, you should not pass true for the useWeakReference argument. For example:

addEventListener("anyEvent",
  function(e:Event) { /* My listener function. */ },
  false, 0, true);

In this example, passing true as the last argument can lead to unexpected results. To Flex, an inner function is actually an object, and can be freed by the garbage collector. If you set the value of the useWeakReference argument to true, as shown in the previous example, there are no persistent references at all to the inner function. The next time the garbage collector runs, it might free the function, and the function will not be called when the event is triggered.

If there are other references to the inner function (for example, if you saved it in another variable), the garbage collector will not free it.

Regular class-level member functions are not subject to garbage collection; as a result, you can set the value of the useWeakReference argument to true and they will not be garbage collected.
Removing event handlers

It is a good idea to remove any handlers that will no longer be used. This removes references to objects so that they can be cleared from memory. You can use the removeEventListener() method to remove an event handler that you no longer need. All components that can call addEventListener() can also call the removeEventListener() method. The syntax for the removeEventListener() method is as follows:

```
componentInstance.removeEventListener(event_type:String, listener_function:Function, use_capture:Boolean)
```

For example, consider the following code:

```
myButton.removeEventListener(MouseEvent.CLICK, myClickHandler);
```

The `event_type` and `listener_function` parameters are required. These are the same as the required parameters for the addEventListener() method.

The `use_capture` parameter is also identical to the parameter used in the addEventListener() method. Recall that you can listen for events during all event phases by calling addEventListener() twice: once with `use_capture` set to true, and again with it set to false. To remove both event listeners, you must call removeEventListener() twice: once with `use_capture` set to true, and again with it set to false.

You can remove only event listeners that you added with the addEventListener() method in an ActionScript block. You cannot remove an event listener that was defined in the MXML tag, even if it was registered using a call to the addEventListener() method that was made inside a tag attribute.

The following sample application shows what type of handler can be removed and what type cannot:

```xml
<?xml version="1.0"?>
<!-- events/RemoveEventListenerExample.mxml -->
initialize="createHandler(event)"
><mx:Script><![CDATA[
    import mx.controls.Alert;
    private function createHandler(e:Event):void {
      b1.addEventListener(MouseEvent.CLICK, myClickHandler);
    }
    private function removeMyHandlers(e:Event):void {
      /* Remove listener for b1's click event because it was added
       with the addEventListener() method. */
      b1.removeEventListener(MouseEvent.CLICK, myClickHandler);
    }
</mx:Script>]]>
```
/* Does NOT remove the listener for b2's click event because it was added inline in an MXML tag. */
b2.removeEventListener(MouseEvent.CLICK, myClickHandler);
}
private function myClickHandler(e:Event):void {
    Alert.show("The button was clicked.");
}
]]><mx:Script>

<mx:Button id="b1" label="Click Me"/>
<mx:Button label="Click Me Too" id="b2" click="myClickHandler(event)"/>
<mx:Button label="Remove Event Listeners" id="b3" click="removeMyHandlers(event)"/>
</mx:Application>

Creating event handler classes

You can create an external class file and use the methods of this class as event handlers. Objects themselves cannot be event handlers, but methods of an object can be. By defining one class that handles all your event handlers, you can use the same event handling logic across applications, which can make your MXML applications more readable and maintainable.

To create a class that handles events, you usually import the flash.events.Event class. You also usually write an empty constructor. The following ActionScript class file calls the Alert control's show() method whenever it handles an event with the handleAllEvents() method:

// events/MyEventHandler.as
package { // Empty package.
    import flash.events.Event;
    import mx.controls.Alert;

    public class MyEventHandler {
        public function MyEventHandler() {
            // Empty constructor.
        }

        public function handleAllEvents(event:Event):void {
            Alert.show("Some event happened.");
        }
    }
}

In your MXML file, you declare a new instance of MyEventHandler and use the addEventListener() method to register its handleAllEvents() method as a handler to the Button control's click event, as the following example shows:

<?xml version="1.0"?>
<!-- events/CustomHandler.mxml -->
The best approach is to define the event handler’s method as static. When you make the event handler method static, you are not required to instantiate the class inside your MXML application. The following `createHandler()` function registers the `handleAllEvents()` method as an event handler without instantiating the `MyStaticEventHandler` class:

```xml
<?xml version="1.0"?>
<!-- events/CustomHandlerStatic.mxml -->
    <mx:Script><![CDATA[
        private function createHandler():void {
            b1.addEventListener(MouseEvent.CLICK, MyStaticEventHandler.handleAllEvents);
        }
    ]]>></mx:Script>
    <mx:Button label="Submit" id="b1"/>
</mx:Application>
```

In the class file, you just add the `static` keyword to the method signature:

```as
package { // Empty package.
    import flash.events.Event;
    import mx.controls.Alert;

    public class MyStaticEventHandler {
```

```as
    public static function handleAllEvents(event:MouseEvent):void {
        Alert.show("Event event = " + event.type);
    }
```

```as
} // MyStaticEventHandler
```

```as
// events/MyStaticEventHandler.as
package { // Empty package.
    import flash.events.Event;
    import mx.controls.Alert;

    public class MyStaticEventHandler {
```
public function MyStaticEventHandler() {
    // Empty constructor.
}

public static function handleAllEvents(event:Event):void {
    Alert.show("Some event happened.");
}

Store your event listener class in a directory in your source path. You can also store your ActionScript class in the same directory as your MXML file, although Adobe does not recommend this.

Handling events

Flex applications are event driven. Events let a programmer know when the user has interacted with an interface component, and also when important changes have happened in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing.

When an instance of a component dispatches an event, objects that have registered as listeners for that event are notified. You define event listeners, also called event handlers, in ActionScript to process events. You register event listeners for events either in the MXML declaration for the component or in ActionScript. For additional examples of the event handling, see Initializing components at run time.

The following example registers an event listener in MXML that is processed when you change views in an Accordion container.

<?xml version="1.0"?><!-- components\CompIntroEvent.mxml --><mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="300"
    height="280">
<mx:Script>
<![CDATA[
import mx.controls.Alert;
private function handleAccChange():void {
    Alert.show("You just changed views.");
}
]]>
</mx:Script>

<!-- The Accordion control dispatches a change event when the selected child container changes. -->
The executing SWF file for the previous example is shown below:

This example produces the following image:

![Accordion Example](image.jpg)

You can pass an event object, which contains information about the event, from the component to the event listener.

For the Accordion container, the event object passed to the event listener for the `change` event is of class `IndexChangedEvent`. You can write your event listener to access the event object, as the following example shows:

```xml
<?xml version="1.0"?>
    <mx:Script>
        <mx:HandleAccChange();> 
        <mx:HBox label="Box 1">
            <mx:Label text="This is one view."/>
        </mx:HBox>
        <mx:HBox label="Box 2">
            <mx:Label text="This is another view."/>
        </mx:HBox>
    </mx:Script>
</mx:Application>
```
Working with Events in Flex

<!-- Import the class that defines the event object. -->
import mx.events.IndexChangedEvent;
import mx.controls.Alert;

private function handleChange(event:IndexChangedEvent):void {
  var currentIndex:int=event.newIndex;
  Alert.show("You just changed views.\nThe new index is "
          + event.newIndex + ".");
}

<!-- The Accordion control dispatches a change event when the selected child container changes. -->
<mx:Accordion id="myAcc"
  height="60"
  width="200"
  change="handleChange(event);">
  <mx:HBox label="Box 1">
    <mx:Label text="This is one view."/>
  </mx:HBox>
  <mx:HBox label="Box 2">
    <mx:Label text="This is another view."/>
  </mx:HBox>
</mx:Accordion>

The executing SWF file for the previous example is shown below:

In this example, you access the newIndex property of the IndexChangedEvent object to determine the index of the new child of the Accordion container. For more information on events, see Using Events.

About the component instantiation life cycle

The component instantiation life cycle describes the sequence of steps that occur when you create a component object from a component class. As part of that life cycle, Flex automatically calls component methods, dispatches events, and makes the component visible.

The following example creates a Button control and adds it to a container:

<?xml version="1.0"?
<!-- components\AddButtonToContainer.mxml -->
  <mx:Box id="box1" width="200">
    <mx:Button id="button1" label="Submit"/>
  </mx:Box>
</mx:Application>
The following ActionScript is equivalent to the portion of the MXML code. Flex executes the same sequence of steps in both examples.

```actionscript
// Create a Box container.
var box1:Box = new Box();
// Configure the Box container.
box1.width=200;

// Create a Button control.
var button1:Button = new Button()
// Configure the Button control.
button1.label = "Submit";

// Add the Button control to the Box container.
box1.addChild(button1);
```

The following steps show what occurs when you execute the ActionScript code to create the Button control, and add it to the Box container. When you create the component in MXML, Flex SDK generates equivalent code.

1. You call the component's constructor, as the following code shows:
2. // Create a Button control.
3. var button1:Button = new Button()
4.
5. You configure the component by setting its properties, as the following code shows:
6. // Configure the button control.
7. button1.label = "Submit";
8.
9. You call the addChild() method to add the component to its parent, as the following code shows:
10. // Add the Button control to the Box container.
11. box1.addChild(button1);
12.

Flex performs the following actions to process this line:

13. Flex sets the parent property for the component to reference its parent container.
14. Flex computes the style settings for the component.
15. Flex dispatches the add event from the button.
16. Flex dispatches the childAdd event from the parent container.
17. Flex dispatches the preinitialize event on the component. The component is in a very raw state when this event is dispatched. Many components, such as the Button control, create internal child components to implement functionality; for example, the Button control creates an internal UITextField component to represent its label text. When Flex dispatches the preinitialize event, the children (including the internal children, of a component) have not yet been created.
18. Flex creates and initializes the component's children, including the component's internal children.

19. Flex dispatches the initialize event on the component. At this time, all of the component's children have been initialized, but the component has not been fully processed. In particular, it has not been sized for layout.

20. Later, to display the application, a render event gets triggered, and Flex does the following:

1. Flex completes all processing required to display the component, including laying out the component.
2. Flex makes the component visible by setting the visible property to true.
3. Flex dispatches the creationComplete event on the component. The component has been sized and processed for layout and all properties are set. This event is dispatched only once when the component is created.
4. Flex dispatches the updateComplete event on the component. Flex dispatches additional updateComplete events whenever the position, size, or other visual characteristic of the component changes and the component has been updated for display.

You can later remove a component from a container by using the removeChild() method. The removed child's parent property is set to null. If you add the removed child to another container, it retains its last known state. If there are no references to the component, it is eventually deleted from memory by the garbage collection mechanism of Adobe® Flash® Player.

Given this sequence of actions, you should use the events as follows:

- The preinitialize event occurs too early in the component life cycle for most initialization activities. It is useful, however, in the rare situations where you must set the properties on a parent before the children are created.
- To configure a component before Flex has determined its visual appearance, use the initialize event. For example, use this for setting properties that affect its appearance, height, or width.
- Use the creationComplete event for actions that rely on accurate values for the component's size or position when the component is created. If you use this event to perform an action that changes the visual appearance of the component, Flex must recalculate its layout, which adds unnecessary processing overhead to your application.
- Use the updateComplete event for actions that must be performed each time a component's characteristics change, not just when the component is created.