**Action Message Format and Data-centric Flex**

Events are central to understanding how to work with Flex and how to create compelling, well-designed applications. Events are a powerful tool and one of the main features of the Flash Player.

Events are deceptively simple, but at the same time, they're an extremely powerful communication mechanism for your applications. Flex is built around an event-driven framework. Events are the central nervous system within your applications, with information flowing in and out of them, up and down, and side to side. We don't mean to be melodramatic, but without understanding events Flex becomes a terrible chore and ActionScript 2.0 begins to look sexy again.

Let’s begin by introducing the event system and describe its all-important role in a Flex application. Next we will explore sending and receiving events.

**The event system**

When working with web technologies such as ColdFusion, .Net, ASP, PHP, PERL, and Python, information is transmitted on a request and response basis or what is called synchronous requests. In this model, remote servers wait for requests from clients (users), then gather and process information relevant to the request. They build the appropriate response and send the data back to the client, at which point the transaction is over—until the client requests the next transaction. Figure 1 shows a request from your laptop, over the Internet, to the web server (where the site is held), and back to your computer. Between step 1 and 2 there is latency, or a delay, due to the time it takes to travel from your location to the location of the server. The same goes for step 3 to step 4.
The Flash Player employs a different paradigm. Instead of sending out requests and receiving responses, the Flash Player makes requests then listeners patiently wait and listen for asynchronous events. When a listener hears an event, it performs the task it was designed to do then waits again for the next event to occur.

Figure 2 shows an example of a profile manager with a form and the resulting information as two separate components. In phase 1, the top graphic, the “Your Information” pod registers as a listener for “update” events from the “Update your information” component. At this point nothing happens until the update event is dispatched and the rest of the application is still available for interaction. Once the user enters some information and clicks the “Update” button, an event is dispatched and anything listening for this event will be notified; as shown phase 2, the bottom graphic. Notice all of this interaction happens within the application and never returns to the server.
In the Flash Player, events are constantly fired in response to a variety of user inputs and system notifications. The main instigator for these events is the user. By clicking a mouse button, moving the mouse, or selecting an item from a drop-down menu, the user is unwittingly setting off events, which trigger the application to respond accordingly.

With Flash Player applications, the difference is in the user-transparent operations that take place in the background. This gives you the ability to fire off tasks asynchronously, in parallel, none of which are dependent on each other, allowing for transparent activities to go on behind the scenes while your application interacts with and provides feedback to the user in real time (versus having code execute synchronously, or sequentially, while the user waits for it to complete).

In addition, when requests for new information are made, they’re sent to the target server and the application carries on with other tasks or patiently waits. The response will be processed whenever the result comes back, whether that happens to be in five seconds or five minutes.

In contrast, when making a request to a web server, an HTML page viewed using a web browser shows the page in a vertical manner starting with the top of the page to the bottom of the page, unless JavaScript is used to asynchronously update the page after the page has loaded.

We’ll get into how this works in a moment; before we do, let’s look at how the Flash Player event system is similar to another system you already know well, “I’ll call you, don’t call me.”

**Event system—the Hollywood Principle**

Components and classes, also known as dispatchers or targets, need to communicate and pass data to each other through the event system. To illustrate this more clearly, we can draw a parallel between the event system and the Hollywood Principle, which states, “Don’t call me, I’ll call you.”

Think of a person calling you asking to borrow money. You tell them you will contact them with an answer after checking your bank account first. This is the Hollywood Principle which is nothing more than an asynchronous request in tech terms. Consider the following example.

Your Flex application is supposed to load data from the server but while the data is loaded you want to show some cute, animated message to the user. The data loading component would be you on the phone, the application is the person in need, and the cute, animated message would be you dancing to Footloose in your mirror while checking your bank account.

The request has been made. The amount of time between the request and the resulting call back is undetermined. So, you could be dancing for a while, but once you have checked your account and have an answer you are ready to inform the caller of your answer.

This is exactly how Flex applications are developed and why events are critical: dispatchers receive requests and dispatch events accordingly but not necessarily immediately.

**Event-delivery system**

It’s important to know the pathway events take within a Flex application, because this pathway determines which components receive the events and which don’t.

Events originate from the dispatcher, traverse the display tree vertically to the Application root then to the stage, and are sent back down to the dispatcher, as shown in figure 1.
The event goes through the parent tree (and any components that are specifically listening to it), which has implications for which components receive notifications about events. For example, as shown in figure 3, a component’s parent typically receives event notifications; children and siblings don’t receive notifications. From an application perspective, when a component dispatches an event, that event can either bubble or not bubble. If the event bubbles, it traverses up the parent chain to the application root, passing by every parent in the chain. Each parent can listen for application events at its own level and rebroadcast those events as needed, stop the propagation, or call methods to take specific action.

This is the real power behind the Flash Player event system: the ability to create custom events and pass them around. Tying into the event system allows for maximum decoupling of logic and maximizes the components that can use that logic, which in turn affords maximum code reuse.

Now that you’ve learned a bit about how the application passes events around, let’s break down the event’s journey from start to finish by exploring sending and receiving events.

**Sending and receiving events**

A Flash Player event is made up of the following core properties (see table 1).

**Table 1 Core event properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event.target</td>
<td>Event dispatcher</td>
</tr>
<tr>
<td>Event.currentTarget</td>
<td>Component currently containing and is inspecting the event or the dispatcher.</td>
</tr>
<tr>
<td>Event.type</td>
<td>A string name that identifies the type of event, such as a click event (clicking a button), a mouse event (moving the mouse), or a select event (selecting an item). Events come in many types, and each type includes its own unique items; but each event has the generic types mentioned here.</td>
</tr>
<tr>
<td>Event.eventPhase</td>
<td>Current phase of the event</td>
</tr>
</tbody>
</table>
NOTE
In the Target phase `currentTarget` has the same value as `target`, the dispatcher, but in the Capture and Bubbling phases `currentTarget` is different.

Each phase of the event journey—from dispatcher to parents to stage and back again—can be divided into phases. Events have only three phases, depending on where they are in the process, as shown in table 2.

Table 2 The different event phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capture</td>
<td>Event travels from the stage through the parents to the dispatcher</td>
</tr>
<tr>
<td>Bubbling</td>
<td>Event travels from the dispatcher through the parents to the stage</td>
</tr>
<tr>
<td>Target</td>
<td>Occurs only when the event has reached its target object or the dispatcher and only relates to the one object, target or dispatcher</td>
</tr>
</tbody>
</table>

NOTE
Bubbling and Capturing phases travel through parents but never through the children of the dispatcher. You can determine which phase you’re in by using the event object’s `eventPhase` property. This property contains an integer that represents one of the following Event constants:

- `Event.CAPTURING_PHASE:uint = 1`
- `Event.AT_TARGET:uint = 2`
- `Event.BUBBLING_PHASE:uint = 3`

BEST PRACTICE
When you’re referring to or monitoring these phases, you can use either the number or the constant, although it’s considered best practice to use constants wherever possible to make your code easier to read and manage. Other events and custom classes can contain other properties as well, but at minimum they include these properties.

Let’s examine what happens with a simple click event generated by pressing a mouse button, as shown in listing 1.

Listing 1 Event listening with MXML and an ActionScript event listener

```xml
<?xml version="1.0" encoding="utf-8"?>
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      protected function onClick(event:Event):void{
        Alert.show(event.target.label + " clicked", "Event Test");
      }
    ]]>
    <s:Button label="Button 1" click="onClick(event)" />
  </fx:Script>
</s:Application>
```

When you click the button, a click event is generated and the `onClick` listener is called. In the `onClick` listener, we show an alert. The alert is Message is the target items label, `event.target.label`, with " clicked" appended.
NOTE

Remember event.target references the dispatcher so label is a property of the target which, in this case, is a Button component.

Listing 1 uses a script block to manage the click event. This can also be done without a script block. Listing 2 uses inline mxml to show the same alert.

### Listing 2 Event listening with inline mxml

```mxml
<s:Button label="Button 1">
  <s:click>
    <![CDATA[
      Alert.show(event.target.label + " clicked", "Event Test");
    ]]> 
  </s:click>
</s:Button>
```

Listing 2 uses the same button but instead makes the event listening occur inline. This has advantages and disadvantages. Ultimately it boils down to a preference but keep in mind an inline event listener cannot be used by multiple dispatchers. In the case where multiple Button components need to call the same method, this approach would fail miserably.

Although these examples do the event adding in MXML, you are not restricted to work in that environment. You can do the same thing in ActionScript by using the `addEventListener()` function.

### Adding event listeners in ActionScript

Using the `addEventListener()` function provides more fine-grained control over the events and is the only way, in ActionScript, to listen for dispatched events. One huge reason for adding event listeners in ActionScript is event listeners added in MXML cannot be removed.

If you need to listen for an event in any phase other than the bubbling and target phase, you must add the listener using the ActionScript method. As a handy reference guide, we've included the main attributes of an event listener in table 3.

### Table 3 Event listener properties and method arguments

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>String</td>
<td>(Required) Type of event for which to listen. You can define the event type as a string or use best practices and use the event type constant defined on every event object.</td>
</tr>
<tr>
<td>Listener</td>
<td>Function</td>
<td>(Required) The function to call when the event is dispatched.</td>
</tr>
<tr>
<td>useCapture</td>
<td>Boolean</td>
<td>(Optional) Phase in which to listen. If true, the listener listens for the event during the capture phase. The default value is false (uses the bubbling or target phase).</td>
</tr>
<tr>
<td>priority</td>
<td>Integer</td>
<td>(Optional) When the listener is called. The higher the number, the sooner it's called. The value can be negative; the default value is 1.</td>
</tr>
<tr>
<td>weakReference</td>
<td>Boolean</td>
<td>(Optional) How quickly the event listener object is picked up and destroyed by the garbage collector. true means it's discarded sooner. The default value is false, which prevents garbage collection from destroying the listener (performance at the cost of memory).</td>
</tr>
</tbody>
</table>

**WEAKREFERENCE**

weakReference is a hot topic in the ActionScript community pertaining to the proper way of managing application performance and memory. A lot of developers rely heavily on using weakReference while others rely on removing all listeners. The arguments on both sides are solid and it all boils down to preference. Find out
what works best for you and your applications. The easiest is to start with weakReference then move on to your own desired scheme for cleaning up event listeners since the Flash Player does not provide a mechanism for removing all listeners.

Let’s take the previous MXML application from Listing 1 and use ActionScript to register the listener. Listing 3 shows how you add an event listener on a previously instantiated display object using ActionScript.

**Listing 3 Click event example using ActionScript**

```xml
<?xml version="1.0" encoding="utf-8"?>
    xmlns:s="library://ns.adobe.com/flex/spark"
    applicationComplete="init()"
    <fx:Script>
    <![CDATA[
    import mx.controls.Alert;
    protected function init():void{
        button1.addEventListener(MouseEvent.CLICK, onClick); #A
    }
    protected function onClick(event:Event):void{
        Alert.show(event.target.label + " clicked", "Event Test");
    }
]>>
    </fx:Script>
    <s:Button id="button1" label="Button 1" />
</s:Application>
```

#A Sets onClick to run when click event occurs

As demonstrated in listings 1 and 3, the main difference between the MXML and ActionScript methods for adding a simple event listener involves the use of `addEventListener`.

When the button is clicked, it dispatches a click event which activates and passes the click event to `onClick()`. With the ActionScript method, you need an intermediary function to add the event listener to the button.

In listing 3, this is handled by `init()`, which is called when the Application issues the `applicationComplete` event. This event is the last event dispatched in the Application startup and is called when the Application is fully initialized and added to the display list.

**NOTE**

If you add an event listener for the capture phase, passing `true` as the third parameter of `addEventListener`, of a button’s click event, it listens only during the capture phase. If you need to listen to both the capture phase and the bubbling phase, you must add a second event listener omitting or passing `false` as the third `addEventListener` parameter.

**NOTE**

When adding listeners a serious consideration for `weakReference` is required. If you think the object will ever need garbage collection and you are not going to explicitly remove all event listeners from the object, use `weakReference`. Do not use `weakReference` on local objects (objects created in a method then destroyed) since they garbage collection occurs automatically, which could cause your listeners not to fire if they are garbage collected before your expected event.

You've probably already gathered this, but most actions in Flex have corresponding events for which you can listen by using the event listeners; you can then respond as needed. This is the communication and nervous system of your Flex application. Even setting variables can cause events to be broadcast. This type of event dispatch is called `binding`.

For source code, sample chapters, the Online Author Forum, and other resources, go to [www.manning.com/ahmed2/](http://www.manning.com/ahmed2/)
**Binding events**

Binding in Flex is carried out in the event system. When you bind a variable, you’re establishing a dedicated listener that picks up on change events issued from the variable or object to which it’s bound.

Whenever you create a binding to a variable, you register an event listener to respond to any changes that occur in that variable. When binding in MXML, the updating takes place behind the scenes, as demonstrated in listing 4.

### Listing 4 MXML binding

```xml
<?xml version="1.0" encoding="utf-8"?>
   xmlns:s="library://ns.adobe.com/flex/spark">
   <s:layout>
      <s:HorizontalLayout />
   </s:layout>
   <fx:Script>
      <![CDATA[
         [Bindable] #A
         protected var _labelText:String = "Label before event";
      ]]>  
   </fx:Script>
   <s:Button id="myButton" label="Change Label!">
      <s:click>
         <![CDATA[
            _labelText = "Label " + Math.round(Math.random()*10);
         ]]>  
      </s:click>
   </s:Button>
   <s:Label id="myLabel" text="{_labelText}"/>
</s:Application>
```

#A [Bindable] metadata means watch this variable

Compare the code in listing 4 to what is required to accomplish the same thing in ActionScript (listing 5). This ActionScript version relies on a class called the `ChangeWatcher`, which monitors any changes in the value of a property to which you have it bound. If a change occurs, `ChangeWatcher` triggers the necessary events to watch that value. It’s much like an event listener object in that it listens for specific events from a property.

### Listing 5 ActionScript binding using ChangeWatcher

```xml
<?xml version="1.0" encoding="utf-8"?>
   xmlns:s="library://ns.adobe.com/flex/spark"
   applicationComplete="init()">
   <s:layout>
      <s:VerticalLayout/>
   </s:layout>
   <fx:Script>
      <![CDATA[
         import mx.events.PropertyChangeEvent; #A
         import mx.binding.utils.ChangeWatcher; #A
         protected var _watcher:ChangeWatcher;
         protected function init():void{
            toggleWatch();
         }
         protected function toggleWatch():void{
            if(_watcher && _watcher.isWatching()){ #C
               _watcher.unwatch(); #D
               toggleButton.label = "Watch";
            }else{
               _watcher = ChangeWatcher.watch(inputField, "text", onTextChange);#B
               toggleButton.label = "Stop Watching";
            }
         }
         protected function onTextChange(event:Event):void{

```
myLabel.text = inputField.text;
}
]]>
</s:Application>

#A Necessary imports to dynamically bind variables
#B Tells ChangeWatcher to detect changes
#C Checks list of watched variables
#D Removes watched variable

This method isn’t as easy or even the shortest ActionScript approach, but it’s more flexible. Upon receipt of the `applicationComplete` event from the application, you toggle the watcher. Since `_watcher` is null at startup the first `if` statement in `toggleWatch` fails causing the `else` block to run. The `else` block binds to changes on the `inputField`'s `text` property by calling the `ChangeWatcher.watch()` method, which acts as the "factory" method for `ChangeWatcher`; an instantiated `ChangeWatcher` object is returned when you call the `watch()` method.

This method takes inputs for the object you want to watch and a property of the watched object that is listened to (in this case, you’re watching the `text` property of `inputField`). The third property specifies which function to call when this event is triggered—in this case, `onTextChange()`.

If you look back at the `addEventListener()` method, you’ll see this approach acts in a similar manner. With the `addEventListener()` method, you’re listening to the entire object; with the method presented in listing 5, you’re watching a specific object property. Changes made to the object property trigger the event listener.

When you type anything in the text input, `ChangeWatcher` automatically listens for those updates and executes the listening method. Each time you press a key, you send out an event that is monitored by the `ChangeWatcher`. As demonstrated in listing 5, your binding event can be as simple as copying the user input value into the `myLabel` component, or it can be as complex as you need it to be.

Another benefit of this implementation is the ability to remove a binding from an object. You can’t remove the binding in MXML. As shown in listing 5, using ActionScript, when the `toggleWatch` is called and `watcher.isWatching()` is true, the bind event is removed using `_watcher.unwatch()` causing future property changes to not call the event listener.

If you look at the toggle method, the `toggleButton`’s label is changed to reflect the current watch state of `watcher`. This could be done using multiple buttons or by listening to other types of events as well. This approach is just for user interface simplicity.

**NOTE**

`ChangeWatcher.watch()` also has a weakReference argument. Use this argument in the same instances and for the same reasons as you would with `addEventListener()`.

The `BindingUtils` class also allows binding through ActionScript but it is only a wrapper around `ChangeWatcher`. Using `BindingUtils.bindProperty` or `BindingUtils.bindSetter` you can setup a `ChangeWatcher` binding. It is a helper class to condense the lines of code necessary to bind objects in ActionScript. The following line of code is similar to calling `ChangeWatcher.watch()`:

```actionscript
BindingUtils.bindProperty(myLabel, "text", inputField, "text");
```

Table 4 lists a few of the benefits between the different approaches to data binding.

<table>
<thead>
<tr>
<th>Ability to toggle on/off</th>
<th>MXML Binding*</th>
<th>ChangeWatcher</th>
<th>BindingUtils</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Call methods on change</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Two-way binding</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* This includes the simple `{}` and `@{}` syntax as well as using `<fx:Binding/>`. 

For source code, sample chapters, the Online Author Forum, and other resources, go to  
[www.manning.com/ahmed2/](http://www.manning.com/ahmed2/)
Removing event listeners

You've just seen the `unwatch()` method in action, which lets you stop monitoring a variable for changes. When using event listeners, you have the same type of capability. If an event listener was added at runtime in ActionScript, you're able to remove it using the `removeEventListener()` method. In fact, `unwatch()` merely uses `removeEventListener` to stop the events from dispatching. If you look at the underlying code for the ChangeWatcher class, you'll notice on line 500 the following statement:

```
host.removeEventListener(p, wrapHandler);
```

Take a look at listing 6 (testingForListeners.mxml) to see how to remove event listeners.

Listing 6 Example of adding and removing events

```xml
<?xml version="1.0" encoding="utf-8"?>
               xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      protected function toggleListeners():void{
        if(box.hasEventListener(MouseEvent.CLICK)){
          log("Listeners removed");
          box.removeEventListener(MouseEvent.MOUSE_MOVE, onEvent); #A
          box.removeEventListener(MouseEvent.MOUSE_OUT, onEvent); #A
          box.removeEventListener(MouseEvent.MOUSE_OVER, onEvent); #A
          box.removeEventListener(MouseEvent.CLICK, onEvent); #A
        }else{
          log("Listeners added");
          box.addEventListener(MouseEvent.MOUSE_MOVE, onEvent); #B
          box.addEventListener(MouseEvent.MOUSE_OUT, onEvent); #B
          box.addEventListener(MouseEvent.MOUSE_OVER, onEvent); #B
          box.addEventListener(MouseEvent.CLICK, onEvent); #B
        }
      }
      protected function onEvent(event:Event):void{ #C
        log("Event triggered: "+event.type);
      }
      protected function log(text:String):void{
        logField.text = text + "\n" + logField.text;
      }
    ]]>}
  </fx:Script>
  <s:Button label="Toggle Listeners" click="toggleListeners()" />
  <s:Group id="box">
    <s:Rect width="200" height="50">
      <s:fill>
        <s:SolidColor color="0x979797"/>
      </s:fill>
    </s:Rect>
  </s:Group>
  <s:TextArea id="logField" width="400" height="400"/>
</s:Application>
```

#A Removes event listeners
#B Adds event listeners
#C Event listener

Listing 6 shows how to test an object to determine if a particular event listener was added then removes the listeners or adds them back accordingly so the same listener isn’t added multiple times. Run the code and you will see output similar to figure 4.

For source code, sample chapters, the Online Author Forum, and other resources, go to

www.manning.com/ahmed2/
NOTE

removeEventListener() in listing 6 only uses two arguments but it is important to note the third parameter. The third argument is useCapture which is false by default. When an event listener is added for the Capture phase to remove it you must pass true to as the third argument.

Figure 4 Output from listing 6

toggleListeners() introduces a new method: hasEventListener(type:String):Boolean. This method checks the target object for existence of the passed in event type. If the listener hasn’t been added to the target object, the event listener is added; otherwise, it is removed essentially toggling the existence of the listeners. This is for logging purposes; we add and remove four different types of events, allowing you to see the different events trigger accordingly.

Removing event listeners added using the MXML

It's important to remember removing event listeners only works on events added using the ActionScript method of defining an event listener; listener functions added using the MXML format are permanently attached to the object. For example, you can't remove an event listener added using the MXML script shown in the following snippet:

<s:Button label="Toggle Listeners" click="toggleListeners()"/>

If you think you might need to remove the event listener at some point, use the addEventListener() method.

Knowing how to add and remove event listeners is the starting point to building a great application.

For source code, sample chapters, the Online Author Forum, and other resources, go to www.manning.com/ahmed2/
Here are some other Manning titles you might be interested in:

- **Hello! Flex 4**  
  Peter Armstrong

- **Flexible Rails**  
  *Flex 3 on Rails 2*  
  Peter Armstrong

- **Flex on Java**  
  Bernerd Allmon and Jeremy Anderson

Last updated: May 25, 2011

For source code, sample chapters, the Online Author Forum, and other resources, go to [www.manning.com/ahmed2/](http://www.manning.com/ahmed2/)