
EFFORTLESS
FLEX 4
DEVELOPMENT
LARRY ULLMAN

Effortless Flex 4 Development
Larry Ullman

New Riders
1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)
Find us on the Web at: www.newriders.com
To report errors, please send a note to: errata@peachpit.com

New Riders is an imprint of Peachpit, a division of Pearson Education.

Editor: Rebecca Gulick
Production Coordinator: Myrna Vladic
Compositor: Debbie Roberti
Copy Editor: Elle Yoko Suzuki
Proofreader: Patricia Pane
Cover and Interior Design: Terri Bogaards
Indexer: Valerie Haynes Perry
Technical Editor: Ryan Stewart

ISBN 13: 978-0-321-70594-5
ISBN 10: 0-321-70594-7
9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

CONTENTS

INTRODUCTION

The Rich Internet Application (RIA) has been a driving force online for the past several years. RIAs provide an improved user experience compared to what's possible using HTML alone and the somewhat clunky interface that is the Web browser. A good RIA turns the Web experience into something much closer to what users have become accustomed to with desktop applications: easy to use, without too many overt server requests, and clearly the work of a professional Web developer.

RIAs can be created using a handful of technologies, but the top two choices are certainly Flash and Ajax ("Ajax" being the name often given to the combination of HTML and JavaScript). People commonly think of Flash as an animated, timeline-driven thing designers use to create games and advertisements, but there's an entirely different side to Flash, based upon the Flex framework.

This book is intended as a programmer's guide to Flex. As I have practically no design skills, you won't see the *best-looking* Flash content, but you will discover and master the best techniques for creating *functional* Flash content. In short, this book is the perfect guide to learning an excellent technology for making today's exemplary Web applications.

WHAT IS FLASH?

Flash is talked about a lot, but there's still quite a bit of misunderstanding about what, exactly, Flash is. Flash, in its broadest sense, is a platform of software and technologies created by Adobe for the purpose of presenting dynamic content. The content itself can range from basic text to forms to multimedia and games. In a narrow sense, Flash is another medium for communication, like HTML, PDFs, or video. However you prefer to think of Flash, it all starts with the development tools and ends with the user's environment.

Adobe has created three programs for generating Flash content:

- Flash Professional
- Flash Builder
- Flash Catalyst

Flash Professional is the current version of the original program used to generate Flash. It provides graphic designers a way to create content that plays out over time, like animation and games. In many ways, Flash Professional is what people generally associate with Flash: a tool for designers.

Flash Builder was named Flex Builder in previous versions.

Flash Builder is an Integrated Development Environment (IDE) for creating Flash content using the Flex framework. Flex will be discussed next in the chapter, but just know that Flex provides a programmatic, event-driven approach to Flash development. Just as Adobe's Dreamweaver application makes Web development faster and easier, Flash Builder makes Flex development faster and easier. I will add that although the book discusses Flash Builder, it really focuses on Flex itself, so you aren't required to have Flash Builder to follow along.

Flash Catalyst is a new product in the Adobe family and acts as an agent between graphical applications like Flash Professional or even Photoshop and the programming tool Flash Builder.

No matter how you develop Flash content, the output will be a Flash file, with an *.swf* extension (pronounced "swiff"). Flash content can then be run in one of three ways:

- In a Web browser that has a Flash Player plug-in installed
- Outside of a Web browser, using the standalone Flash Player
- Outside of a Web browser, using Adobe AIR

Adobe AIR provides a way to create desktop applications using Web technologies.

Generally speaking, Flash content run in a Web browser through the Flash Player plug-in is the most common use of Flash. And, towards that end, most of this book is written with that in mind, although you'll encounter some information about developing for Adobe AIR, too.

WHY USE FLASH?

Flash is one way to present dynamic content but certainly not the only way. Whether you're making an interactive Web site or a standalone desktop application, you have your choice of technologies to use. But before making a comparison, let's look at the argument for Rich Internet Applications as a whole.

The Case for RIAs

Over the course of the Internet's relatively short history, it has already gone through several unique phases. At first, the Web used just HTML and images to present information statically. Then, JavaScript and plug-ins added the ability to run animations, display video, and create (often annoying) effects. Splash pages, pop-ups, and blinking text were not the Internet's high point! But static HTML, with or without adornment, is of limited value and far too impractical to maintain. To make a better Web for end users and developers alike, several server-side technologies arose for generating HTML content

on the fly. The most common of these include Microsoft's ASP, Adobe's ColdFusion (originally by Macromedia), Java Server Pages (JSP), and my personal favorite, PHP.

Server-side technologies made dynamic sites possible, but still relied upon the classic client-server model, where the client (aka, the browser) makes a request of the server, the server returns its response, and the client redraws itself accordingly. For each user action, a separate request and another redrawing of the browser was required. It was time for something new...

Over time, through technologies such as Flash and Ajax, the client-server model has been significantly altered. What Flash and Ajax have in common is that both can make server requests behind the scenes, unbeknownst to the user, and then update the client in a more seamless manner. The end result is a better user experience, akin to that of a desktop application. Some call this "Web 2.0" (I prefer not to), but whatever you call it, the fact is that the best, most popular, and certainly the most useful Web sites around, all qualify as Rich Internet Applications.

The Case for Flash

Why should you use Flash to develop your next RIA then, instead of HTML and JavaScript, the most logical alternative? First of all, thanks to Flex, developing Flash applications can be really quick. And I mean really, really quick. With a fair understanding of Flex, you can complete projects in a day that would take you a week to do using HTML and JavaScript alone. And as your RIAs become more complex, this is even truer. Achieving simple effects and interactivity using JavaScript is not that hard, but anything of moderate to advanced complexity gets to be really difficult to do in JavaScript, even if you use a framework.

A second reason to use Flash is that the content will look and function the same in all browsers regardless of type, version, or operating system. If you've done any Web development at all, you know that getting a site to look and work the same in various browsers can be a tedious chore: in Internet Explorer 6, 7, and 8 on Windows; Firefox and Safari on Windows and other operating systems. Flash will always run through the Flash Player plug-in, so there's a consistent experience regardless of the browser or operating system in use.

But what if the user doesn't have the Flash Player plug-in installed? That would actually be surprising, as there's a nearly 98% adoption rate of the plug-in! And if the user doesn't have a current enough version of the player, they'll be prompted to upgrade. Conversely, well-written Ajax sites should degrade smoothly, but the responsibility is on you, the developer, to make sure that's the case. And, again, you have to test the degradation on multiple browsers, with multiple JavaScript settings.

Finally, another benefit to using Flash is that the Flash platform changes quickly with the times and expectations of users. Being a proprietary technology isn't always a bad thing. Adobe regularly improves upon the Flash platform to reflect what people want to be able to do. Conversely, JavaScript evolves over the course of years, not months, so JavaScript developers often have to play catch-up.

The Case Against Flash

It wouldn't be honest or credible to argue why you should use Flash without mentioning the valid reasons not to. First, you'll need to learn a new technology, in this case, Flex. I personally like learning new things, but not everyone wants to be bothered. If you've already mastered HTML and JavaScript, then maybe you don't have a compelling need to learn Flex. Second, although the Flash Player plug-in is installed on practically every browser, if it's not, then the Flash content will not be shown at all; there is no "degrade nicely" option.

The third reason you may not want to use Flash is because of its incomplete support on mobile devices. While some phone companies are specifically working with Adobe to allow Flash content to be viewed using their devices, Apple is famously resistant, meaning that iPod, iPhone, and iPad users won't be able to see your Flash work. (As I write this, Apple and Adobe are in the middle of a public fight over this very issue, but this could change in time, too.)

Finally, although you can develop Flash content without spending a dime (without even using any of Adobe's tools, in fact), you're likely to spend some money if you embrace the Flash platform at all. For example, the Flash Builder application is excellent, but is a commercial product (see Adobe's site for specific pricing).

All that being said, if you're looking at this introduction, then I assume you're at least curious about Flex (and therefore Flash), so I'd recommend you at least give it a quick spin around the block. This book shouldn't set you back much, Flex is an open source technology, and Flash Builder is available in a 60-day free trial, so you can decide for yourself if it is worth your while, without a significant financial investment.

note

As a minor point, printing Flash content is less than ideal, but I personally hardly ever consider printing out online content.

WHAT IS FLEX?

note

The first chapter discusses Flex, MXML, ActionScript, and Flash Builder in more detail.

As I already said, Flex is a framework for creating Flash content. Most people think of Flash as a graphic design technology, and think they need those skills for Flex, but Flex is for programmers. Trust me on this: I have absolutely no design skills whatsoever, but that hasn't hindered my Flex development at all.

The Flex framework consists of two pieces: MXML and ActionScript. MXML defines the elements you'll use in an application, and the primary application file will be an MXML document. MXML is very similar in both syntax and usage to HTML. ActionScript is an object-oriented programming language used to add logic to an application. ActionScript is very similar in both syntax and usage to JavaScript. But, simply stated, Flex is a framework: an easy means to an end.

If you're using the Flash Builder IDE to develop in Flex, it comes with the Flex framework, along with all the tools needed to turn Flex into Flash (called *compilers*). If you're not using Flash Builder, you'll need to download the free Flex Software Development Kit (SDK). Along with the Flex framework itself, the Flex SDK contains the compilers, code templates, examples, and more.

Flex has been around since 2004 and is currently in version 4, released in 2010. That's the version this book uses exclusively.

Flex 4-generated Flash content will run on Flash Player version 10 and later.

ABOUT THIS BOOK

To understand this book, you should know a little about me. I'm a programmer and developer by occupation, entirely self-taught and with a complete lack of aesthetic skills, as previously mentioned. As with all my books—this is my 18th—it's written under the principle of "If I were learning this subject for the first time, what kind of book would I want?" Largely this means

- Real-world examples
- No fluff, filler, or esoteric examples that you'd never actually use
- Minimal expectations of the reader
- Crystal-clear explanations

Technical writing is largely a matter of making choices about what you need to know and what you don't need to know. Hopefully, I've done a good job of that. By the end of the book you should have a sound understanding not just of how to use Flex but how to use it well. Everything you *need* to know about Flex is covered including, most importantly, how you go about learning more when your knowledge and needs have eclipsed the parameters of this text.

I should also point out that I have a strong aversion to spending any money at all. I say that because I generally use open source software and free applications whenever possible. Still, I find Flex to be a strong enough tool that I use it—and willingly pay for Flash Builder. But this book is not for the purpose of making money for Adobe, and so I'm not going to assume you're necessarily using Flash Builder. The book is really about Flex. Still, Flash Builder is an

excellent tool that most Flex developers are in fact using, so the book does discuss some Flash Builder-specific topics. And now that I'm talking more about the book and less about me, I'll point out that the book doesn't spend much time dwelling on how Flex 4 differs from Flex 3 or earlier versions. If you already know Flex 3, you'll see a few notes that X or Y has changed, but that's about it. Flex 3 is Flex 3 and Flex 4 is Flex 4 and this here book is about Flex 4.

A lot of the talk surrounding Rich Internet Applications goes into how you can create a great user interface. And while that's both true and valid, I'm of the opinion that the *data* used by an RIA is the real star of the show. It's the data that gives users the reason to be at a Web site in the first place and it's the data that will give them reason to return. Accordingly, the meat of the book can be found in Part 2, "Data and Communications." The five chapters there cover everything you need to know about displaying, manipulating, and transmitting data, including three chapters on the client-server relationship. For those chapters, I exclusively use PHP as the server-side technology. Although I explain the purpose of every line of PHP code, if you're not familiar with PHP, you may have difficulty with two of those chapters. In such a case, I could recommend my *PHP for the Web: Visual QuickStart Guide* (Peachpit Press, 2009) book, or any of the available PHP tutorials to be found online. Similarly, MySQL will be used as the database application in those chapters.

So what else do I expect of you, the benevolent reader? I expect many readers will be like me, having used PHP and MySQL but now accepting the need to also know a powerful client-side technology. I expect you probably have done some Web development, having dirtied your hands with HTML even the slightest. If this is true and you have even minimal familiarity with JavaScript, learning Flex will not be a problem. Flex is easy to learn and not even that hard to master.

If you do have problems along the way, you can turn to the book's corresponding Web site: *www.DMCInsights.com/flex4/*. There you can download all of the code from the book, find supplemental material (I also write about Flex, PHP, and other topics on my blog), and get assistance through my online forum.

4 EVENT MANAGEMENT

If you've been reading this book sequentially, you should already have a pretty sound sense of the Flex landscape. The first chapter shows how to create applications. The second introduces the basic elements of a user interface. And the third teaches the fundamentals of ActionScript programming. But in order for applications to be truly a user experience (which is to say, *interactive*), you need to know how to manage events. A lot of what applications do is watch for, and respond to, events.

You can develop Flash content using Flash Professional, the old standard, or Flash Builder (previously known as Flex Builder). Flash Professional tends to start with animation that runs over a timeline. Conversely, Flex development is event-driven: The application is told to respond to things that happen. This take may be different than what you've done before, but it's really easy to adopt. If anything, the biggest hurdle will be making the most of what's possible.

Flex uses an event system that's quite close to the Document Object Model (DOM) Level 3 Event Specification present in Web browsers (to varying degrees). What this means is that if you've done a wee bit of JavaScript programming, much of the syntax and theory in this book will be familiar. And even if you haven't, you may be pleasantly surprised to see how obvious much of this information is. For example, can you guess what event represents the cursor going over an element? Yes, *mouseOver*.

The chapter begins with several pages discussing the premise of event management and what pieces are involved. Then you'll learn how to handle events by placing ActionScript code within MXML components. The third section of

the chapter walks through using functions to handle events (the functions are called *event handlers* in such circumstances). After that, I discuss the types of events in a bit more detail. The chapter concludes with an alternate way to manage events in an application.

FUNDAMENTAL CONCEPTS

Event-driven development can be summed up as follows: When *this event* occurs with *this thing*, take *these actions*. Just a few examples are

- Fetch some data from a server after the application loads.
- Change the choices in one drop-down menu after the user makes a selection in another (e.g., one drop-down represents car makes, the other car models).
- Reveal a block of information when the user moves the cursor over an image.
- Make new application options available once a network connection is detected.
- Store user-supplied data in a database after the user clicks a button.
- Move some other components around after one is removed.
- Update the contents of a **DataGrid** (a table-like component, introduced in the next chapter) as new information becomes available.

The events are already predefined for you within the framework. Events fall under two categories: system events and user events. System events are not caused by user actions. Specific types include the application being fully loaded (i.e., ready to run), components being created, responses being received (like from a network connection), and so forth. User events are the direct result of a user action: moving the cursor, typing in a text input, selecting from a drop-down menu, checking a check box, clicking a button, and more.

The things involved are the application's elements: its controls, containers, even the root application itself. The actions to be taken are defined by you, and represent the greatest range of possible options. This is where you would decide to update a drop-down menu, show a block of information, and so forth.

Once you've identified the event you want to watch for and the element it should be associated with, there are two ways of telling the application what

actions to take when that event occurs. You can place your instructions inline or in a function. Chapter 3, "ActionScript You Need to Know," demonstrates both. The following button gets moved to the right ten pixels every time it is clicked:

```
<s:Button id="myButton" x="20" y="20" click="myButton.x += 10"
label="Click Me!" />
```

The inline ActionScript, used as the value for the *click* property, does the work.

Chapter 3 also has examples of the *click* event calling a user-defined function:

```
<fx:Script>
<![CDATA[
private function moveMe( ):void {
    myButton.x += 10;
}
]]>
</fx:Script>
<s:Button id="myButton" x="20" y="20" click="moveMe( );" label="Click
Me!" />
```

Again, this is all there really is to event-driven programming: Tell the application that when *this event* occurs with *this thing,* take *these actions*. Establishing this connection in your application results in *event handlers*, also called *event listeners*.

Before moving on, there are two more things to understand about events in Flash and Flex. First, the events themselves are going to occur whether you address them or not. Say you have a **VGroup** that has not been instructed to respond to a mouseover event:

```
<s:VGroup>
</s:VGroup>
```

When the user moves the cursor over this **VGroup**, the mouseover event still occurs, but nothing happens as a response.

It's also important to know that Flex events are *asynchronous*, which means that they don't have to wait for each other, or their responses. In other words, multiple events and multiple event reactions can, and often will, take place simultaneously. For example, while a user may be moving the cursor over a component, the application itself may also be handling the data sent back from a server request.

The Event Object

ActionScript is an object-oriented language, which means that most everything you work with in Flex, from components to strings, is an object. This includes events, as well. ActionScript has a generic **Event** class that defines much of the functionality needed to work with events. From this root class, there are many children that inherit from **Event**; each child being a more specific type of event. Every time an event occurs within a Flash application, some type of object from the **Event** lineage is created.

The **Event** family of objects, like any other, has properties and methods for you to use. Here are the three key properties:

- **type**
- **target**
- **currentTarget**

The **Event**'s **type** property reflects what kind of event just happened. This may be click, initialize, mouseover, change, etc. The actual values will be represented by constants like **MouseEvent.CLICK**.

The **target** property of **Event** is an object reference to the component that generated the event. If you click a **CheckBox** with an **id** of *someCheckBox*, the **target** of that click event will be **someCheckBox**. This means that by referencing the **Event**'s **target** property, you can get to the properties and methods of that target object. This will mean more later in the chapter.

The **currentTarget** property—as well as two others, **bubbles** and **eventPhase**—comes into play when dealing with the flow of events, to be covered next.

More specific event types add new properties and methods. For example, the **MouseEvent** object has an **altKey** property that returns a Boolean value indicating if the ALT key was pressed when the mouse event took place. When you go to handle events, you have the option of working with the generic **Event** object or a specific event object. You'll see this later in the chapter.

You'll want to familiarize yourself with the ActionScript documentation for the various events. Start by looking up the **flash.events.Event** class in either the standalone Adobe Help application or online (*http://help.adobe.com/en_US/FlashPlatform//reference/actionscript/3/flash/events/Event.html*, **Figure 4.1** on the next page). Near the top of the page are links you can click to go directly to listings of the class's properties, methods, and constants. The top of the page also shows every class that is derived from **Event** (i.e., subclasses). You can click any subclass name to see the documentation for that class.

tip

You can investigate the properties and values of an event, like any other variable, using Flash Builder's Debug perspective, specifically the Variables window. See the end of Chapter 3 for an introduction to this.

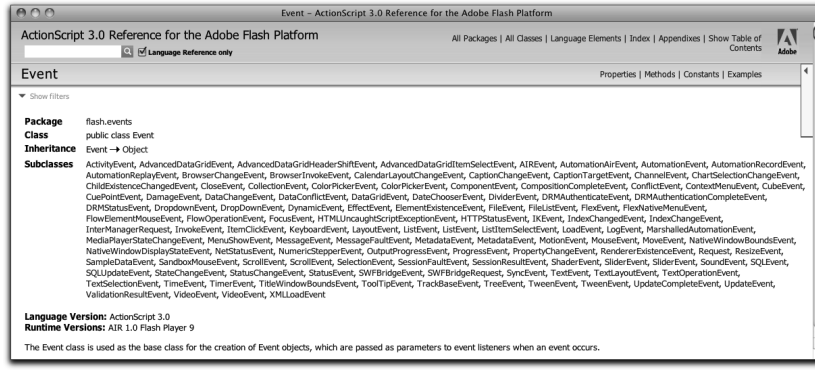

Figure 4.1

The second thing you'll often want to do within the ActionScript documentation is view the events that can be triggered by any given component. If you load the reference for a component, like **Label** in **Figure 4.2**, you'll see an *Events* link near the top of the page after *Properties* and *Methods*. Clicking that link takes you to the full listing of events that the component supports. There are several dozen possible events just for the **Label** component, and all that component does is display a bit of text!

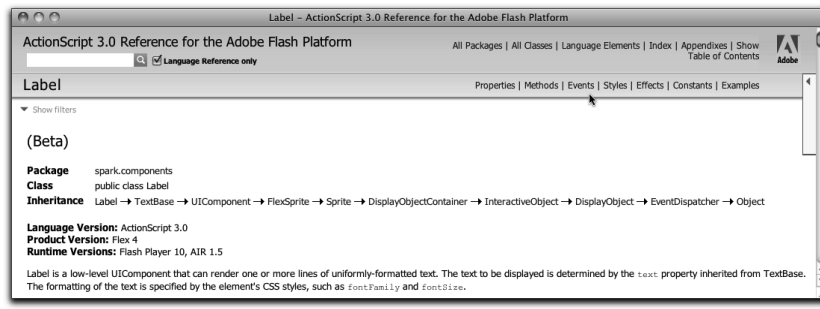

Figure 4.2

Obviously, considering the limitations of a book, I cannot go through uses of every possible event for every possible component (and you wouldn't want that anyway), which is why you'll need to familiarize yourself with navigating the ActionScript documentation. If you're using Flash Builder, you'll see that it provides code hinting for common and available events, too. Events in Flash Builder are represented by a lightning bolt (**Figure 4.3**).

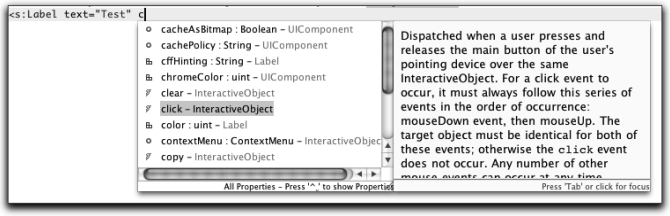

Figure 4.3

Later in the book I'll cover movement-related events (for dragging and dropping), and those having to do with effects. There are also many events specific to AIR application development. These include objects of type **AIREvent**, **FileEvent**, **SQLEvent**, among others. Any property, constant, or method only available in AIR applications are marked with the AIR icon (as in the **Label**'s **contextMenu** event in **Figure 4.4**).

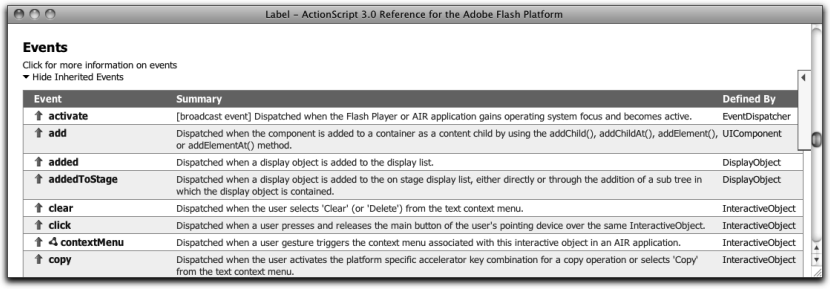

Figure 4.4

Event Flow

The final thing to know about events, before getting into some actual programming, is how events are *propagated* in an application, which is to say how events move about. It's not as simple as just clicking a **Button** creates a click event: That does happen, but so does much more.

To start, let's say there's a **Label** within a **Panel** that's a direct child of the **Application**:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955">
    <s:Panel>
        <s:Label id="myLabel" text="Click Me!" click="doThis();"/>
    </s:Panel>
</s:Application>
```

tip

Event management goes hand-in-hand with creating good user interfaces. Unless you're purposefully creating Easter eggs, you'll need to ensure that what the user can do is successfully communicated. For example, clicking a button is a logical action, but clicking a bit of text, like a ***Label***, is not.

When the user clicks the **Label**, he or she has also clicked the **Panel** and the **Application**. The program then needs to figure out how that event should be handled. To do so, the event goes through three phases looking for event-handler assignments (**Figure 4.5**):

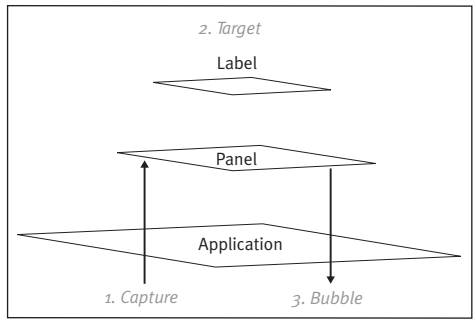

Figure 4.5

1. Capture
2. Target
3. Bubble

In the capture phase the program will start looking for event handlers from the outside (or top) parent to the innermost one. The capture phase stops at the parent of the object that triggered the event. So in my example, the capture phase starts at the **Application** and ends at the **Panel**. If either component has an associated event handler, it will be triggered during this phase (and, consequently, before the **Label**'s event handler).

By default, the capture phase is disabled as it's not commonly used and it can take its toll on the application's performance. But you can define event handlers to watch the capture phase when needed. You might do so to handle an event in a parent component, then prevent it from being handled within a child. Towards the end of the chapter I'll discuss this less common approach to event management.

The second phase, target, is the most important phase. Here the actual subject of the event (i.e., the component that triggered it) is checked for an event handler. This is where most event handling takes place.

Finally, the bubble phase is like capture in reverse, working back through the structure, from the target component's parent on up. There are a couple of reasons you may add handlers to parent components that will catch events during the bubble phase:

- To execute additional actions, besides those triggered by the target phase.

- To execute the same actions when an event is triggered by any of a component's children.

For example, if you have a form with multiple components, you may want to validate an individual component when that component's value changes, and also do something with the entire body of form data at the same time.

The event object's **eventPhase** property stores a number, also represented by a constant, that reflects the current stage (**Table 4.1**).

Table 4.1 Event Phase Values and Constants

Value	Constant
1	**EventPhase.CAPTURING_PHASE**
2	**EventPhase.AT_TARGET**
3	**EventPhase.BUBBLING_PHASE**

Do note that just as not every component can trigger every event type, not all events participate in all three phases. The event object's **bubbles** property returns a Boolean indicating if the event participates in the bubbling phase, in particular.

As already mentioned, the event object's **target** property refers to the component that triggered the event. This value will not change over the course of the event flow. In the example I'm discussing, the target would always be the **Label** object (assuming that was what the user clicked).

The event object also has a **currentTarget** property that reflects the object currently being examined for an event handler. This value will change repeatedly over the course of the event flow. In the **Label**-**Panel**-**Application** example, the **currentTarget** would go from **Application** to **Panel** (in the capture phase) to **Label** (in target) to **Panel** to **Application** (in the bubble phase). I'll demonstrate a utility program in this chapter that may help you understand the phases and targets.

INLINE EVENT HANDLING

The first, most direct, way you can handle events is *inline*. To do so, you assign some ActionScript code to the corresponding event property of a component. For example:

```
<s:Label id="myLabel"/>
<s:TextInput id="myText" change="myLabel.text=myText.text"/>
```

Every time the **TextInput** is changed, the **Label**'s **text** property will be assigned the value of the **TextInput**'s text property. This is functionally equivalent to data binding the two components together:

```
<s:Label id="myLabel" text="{myText.text}"/>
<s:TextInput id="myText" />
```

In fact, data binding is just a simple and direct way to place an event handler on a variable (or object property in this example). When that variable's value changes, an event is triggered. The event handler itself updates the component that is bound to that variable.

You should note Flex assumes that the values assigned to *event properties*, like **change**, will be ActionScript, so the curly braces are not necessary. Using ActionScript for the values of *non-event properties,* like **text**, does require the curly braces.

Figure 4.6

Inline event handling can be applied across components, as in the **Label**-**Text Input** example, or on a component to itself. An earlier example had a **Button**'s horizontal location moved 10 pixels each time it is clicked. This next **Button** will be made 10 pixels taller with each click:

```
<s:Button label="Bigger" id="myButton" click="myButton.height += 10" />
```

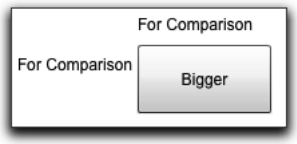

Figure 4.7

If you wanted to increase the **Button**'s height *and* width by 10 with each click, you can add a second ActionScript command after a semicolon (**Figures 4.6 and 4.7**):

```
<s:Button label="Bigger" id="myButton" click="myButton.height += 10;
myButton.width += 10" />
```

While you *can* put multiple commands inline, assigning more than one command to a component property can get ugly pretty quick. Also, you'll find there are limits to what you can do inline. Say you had a check box that, if checked, showed an image; if unchecked, the image should be hidden. Accomplishing that requires more complex logic, best put into a function.

FUNCTIONS AS EVENT HANDLERS

Functions often provide a better way to handle events. Some examples of functions as event handlers are used in Chapter 3. Having a function be called when an event occurs is easy:

- Define the function in a **Script** block.

- Assign the function call as the value of whatever event property in the MXML component.

This approach does a better job of separating your presentation from your logic, allows you to handle events in a more sophisticated manner, and will also let you use the same function to handle all sorts of events on all sorts of components.

tip

The same component can have multiple associated events. A ***Panel*** may watch for mouseover, mouseout, and even click events. Each event needs to be assigned a listener separately, although they can all be assigned the same event handler, when appropriate.

Creating Simple Event Handlers

If you want to create an application that only displays an image if a box is checked (**Figures 4.8** and **4.9**), you can start by defining the components:

```
<s:CheckBox id="showImageCheckBox" label="Show Image?"
change="showHideImage( )" />
<mx:Image id="myImage" source="@Embed('assets/trixie.jpg')"
visible="false" />
```

Figure 4.8

The check box has an **id** value of *showImageCheckBox* and, when changed, calls the **showHideImage( )** function. You'll notice that I'm referencing the *change* event here, because I want the function to be called whenever the check box's status changes, whether that means the user just checked or unchecked it.

Figure 4.9

The image has an **id** of *myImage* and its **source** property will embed the image file at compilation, using a relative path to the image. The image itself is initially invisible (its **visible** property is *false*), meaning it's part of the application but not seen. (In fact, it could even affect the layout while invisible).

Within the **Script** tags, the **showHideImage( )** needs to be defined:

```
private function showHideImage( ):void {
    // Actually functionality.
}
```

The function takes no arguments and returns no values. Within the function, the image's visibility should toggle depending upon whether the check box is selected or not. A simple conditional takes care of that:

```
if (showImageCheckBox.selected == true) {
    myImage.visible = true;
} else {
    myImage.visible = false;
}
```

And that's all there is to it: Check the box and the image appears, uncheck it and the image disappears.

Sending Values to Event Handlers

Event-handling functions can also be written so that they take arguments, just like any function. The value then needs to be passed when the function is called. To do so, you would write that into the component. Here are two **RadioButtons**, each of which calls the same function but sends a different Boolean value to it:

```
<s:RadioButton label="show" click="showHideImage(true);" />
<s:RadioButton label="hide" click="showHideImage(false);" />
```

And here is that function definition:

```
private function showHideImage(visible:Boolean):void {
    myImage.visible = visible;
}
```

The function takes one parameter, of type **Boolean**. Inside the function, the image's **visible** property is assigned the value passed to the function.

Sending Events to Functions

tip

Flash Builder can automatically define the shell of event handlers for you. You'll see this option appear via code completion in Source mode and by clicking the icon of a lightning bolt with a plus sign in the Properties panel of Design mode.

It's probably not obvious why, offhand, but the most likely value you'll want to send to an event-handling function would be an event object. To do so, just define the function call so that it sends along the event variable:

```
<s:Button text="Click Here!" click="myEventHandler(event);" />
```

Then you write the function so that it takes an argument of type **Event**:

```
private function myEventHandler(event:Event):void {
    // Actual functionality.
}
```

By convention, the parameter is named **event** or just **e**.

Within the function, you can make use of the event object's properties and methods. This includes **type**, **currentTarget**, **eventPhase**, and so forth. Most importantly, you can access the object that triggered the event by referencing **event.target**. Any of the target object's available properties and methods are then accessible using **event.target.*whatever***. For example, here's another way of writing the **moveMe( )** function (for moving a component):

```
private function moveMe(event:Event):void {
    event.target.x += 10;
}
```

Instead of hard-coding the component's **id** value in the function, you can refer to the **id** value of **event.target**, where **target** represents the object that

triggered the event. There may be no apparent benefit to this approach, but now that same function can be used on any number of components without a knowledge of what those components are (so long as they have an **x** property):

```
<s:Button id="myButton" x="20" y="20" click="moveMe(event);"
label="Click Me!" />
<s:Label id="myLabel" x="20" y="120" click="moveMe(event);"
text="Click Me!" />
<mx:Image id="myImage" x="20" y="220" click="moveMe(event);"
source="@Embed('assets/image.png')" />
```

Taking this a step further, you could call the same function when any member of the group is clicked:

```
<s:Group click="moveMe(event);">
<s:Button id="myButton" x="20" y="20" label="Click Me!" />
<s:Label id="myLabel" x="20" y="50" text="Click Me!" />
<mx:Image id="myImage" x="20" y="70" source="@Embed('assets/
image.png')" />
</s:Group>
```

Now the same function call applies to the click event in all four objects (clicking anywhere in the group, but not one of the children moves the entire group over ten pixels). **Figure 4.10** shows the application as it originally displays; **Figure 4.11** shows it after multiple clicks.

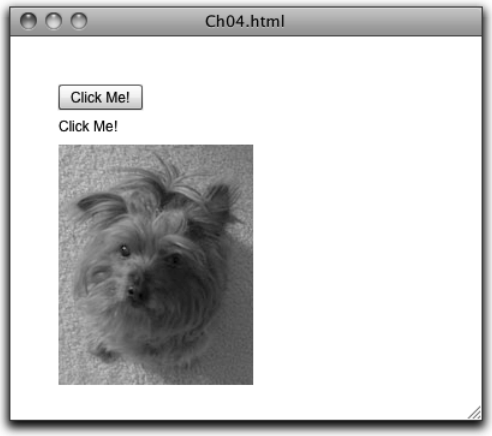

Figure 4.10

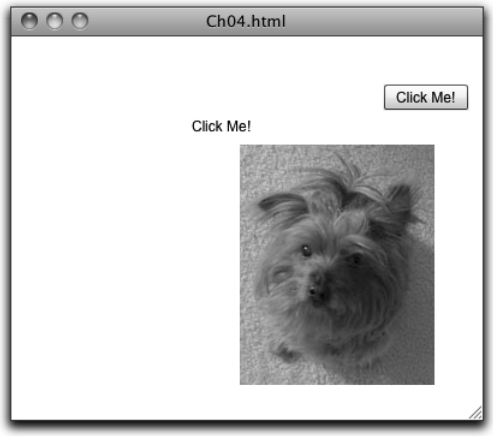

Figure 4.11

Before going further, let's take this information and create a utility for reporting upon what events were triggered by what components. Doing so will better demonstrate the event flow.

tip

If you wanted an event handler to take different actions depending upon the type of component involved, refer to the ***target***'s ***className*** property: ***event.target.className***. You can write a conditional that checks if that property equals ***Button***, ***Label***, etc.

tip

If you download the source code from the corresponding Web site (*www.dmcinsights.com/flex4*), you'll find this code in the **Ch04/Ch04_01** folder.

1. Create a new Flex project for the Web.

2. In the **Application** tag, associate the **reportOnEvent( )** function with any click:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    click="reportOnEvent(event)">
```

   By assigning this event handler in the **Application** tag, the **reportOnEvent( )** function will be called when the user clicks anywhere in the application. The function is passed an event object.

3. Within a **Script** block, define the function:

```
<fx:Script>
<![CDATA[
private function reportOnEvent(event:Event):void {
    results.text += "Target: " + event.target.id + "\ncurrentTarget: " +
    event.currentTarget.id + "\nPhase: " + event.eventPhase +
    "\n----------\n";
}
]]>
</fx:Script>
```

   The function expects one argument of type **Event**. Within the function, the **text** property of the **results** component will be updated. This will be a **RichText** component, added in the next step. Each call of this function will concatenate several strings and values to the current value of the **text** property. Those strings are: *Target:*, followed by a space; the **id** value of **target**; a newline (**\n**, which will space the output over multiple lines), *currentTarget:*, followed by a space; the **id** value of the **currentTarget** property; another newline, followed by *Phase:* and a space; the **eventPhase**, which will be the number 1, 2, or 3; and a newline, followed by several dashes, and another newline. If you're at all confused about how this will look, take a peek at the next three figures in the book.

4. Define the components:

```
<s:VGroup id="myVGroup" paddingLeft="10" paddingTop="10">
    <s:Label id="myLabel" text="Click on Me!" />
    <s:Panel title="Results">
        <s:RichText id="results" />
    </s:Panel>
</s:VGroup>
```

To best demonstrate what's going on, I want to add a couple of components, so I started with a **Label** inside of a **VGroup**. To display the results, I added a **RichText** component within a **Panel**, also in the **VGroup**. **Figure 4.12** shows the application when first loaded.

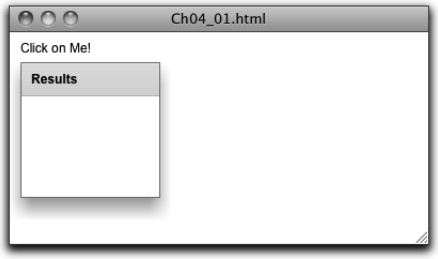

Figure 4.12

5. Save, compile, and run the application.

6. Click the various components, and not any components at all, to see the results (**Figure 4.13**).

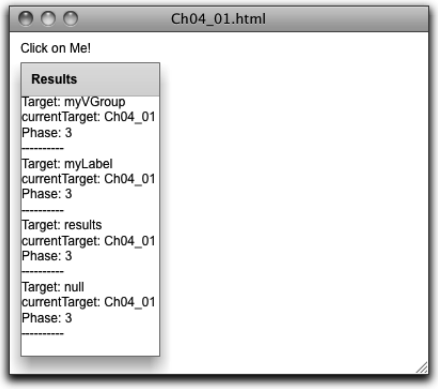

Figure 4.13

The figure shows the results after clicking the **VGroup** first (I clicked to the left of the **Panel**), clicking the **Label** next, the body of the **Panel** (i.e., the **RichText** component) after that, and, finally, outside of any component. You'll note that the **target** for each is the item clicked, except when I clicked outside of any component, in which case the **target** was null. The **currentTarget** is the application (*Ch04_01*) and the phase is 3, bubble, for every click. This is because the only event handler was assigned to the application itself. Let's see what happens when the components get their own event handlers.

7. Add click event handlers to the **VGroup**, **Label**, and **RichText** components:

```
<s:VGroup id="myVGroup" paddingLeft="10" paddingTop="10"
click="reportOnEvent(event)">
    <s:Label id="myLabel" text="Click on Me!"
click="reportOnEvent(event)" />
    <s:Panel title="Results">
        <s:RichText id="results" click="reportOnEvent(event)" />
    </s:Panel>
</s:VGroup>
```

   Each component now also calls the **reportOnEvent( )** function when a click event occurs.

8. Save, compile, and run the application.

9. Click the various components, and not any components, to see the results (**Figure 4.14**).

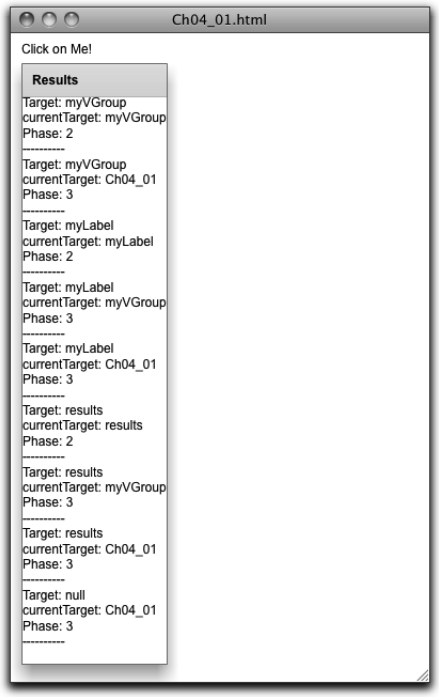

Figure 4.14

   The figure shows the results after clicking the components in the same order as in Step 6: **VGroup**, **Label**, **RichText**, nothing. Now when you click a component, the first event triggered has matching **target** and **currentTarget** values, and occur in the second phase (target). Secondarily, after each target

phase, one or more bubble phase events are triggered, depending upon how nested the target is. For example, clicking the **VGroup** creates one target phase event (when the **VGroup**'s event handler is triggered) and one bubble phase event (when the **Application**'s event handler is triggered). In this latter event, the **target** remains the **VGroup** but the **currentTarget** becomes the application (*Ch04_01*).

You may also notice that no capture phase events are taking place (event phase 1). This is because capture phase event handling is disabled by default. Later in the chapter I'll show you how to change that.

Using Specific Events

As mentioned earlier, it's often best to use specific event object types in your functions rather than the generic **Event**. Doing so requires no change in how the function call is defined:

```
<s:Button text="Click Here!" click="myEventHandler(event)" />
```

But you do need to change the expected parameter type in the handling function:

```
private function myEventHandler(event:EventType):void {
    // Actual functionality.
}
```

The *EventType* value needs to be one of the defined types in ActionScript: **MouseEvent**, **KeyboardEvent**, **DateChooserEvent**, **ToolTipEvent**, etc. Here's how the **moveMe( )** function would be written to take a **MouseEvent** object:

```
private function moveMe(event:MouseEvent):void {
    event.target.x += 10;
}
```

By specifying the event type, you can take advantage of that event's extended abilities. For example, **MouseEvent** has **stageX** and **stageY** properties that reflect where, on the application's stage, the mouse was clicked. You could use that to move an image to the user-designated location:

```
private function moveImage(event:MouseEvent):void {
    myImage.x = event.stageX;
    myImage.y = event.stageY;
}
```

With this particular event handler, it would have to be associated with the application, or a container, and you'd have to reference the image directly by its **id**, as it wouldn't be the **target** of the event (because you wouldn't be clicking the image).

tip

It is possible for a function to take a generic ***Event*** parameter and then typecast that object to a more specific type, but there are fewer justifications for doing so.

Before getting any further, understand that in order for the application to have access to that event type definition, you may need to import the corresponding class or package. Many events, like **MouseEvent** and **KeyboardEvent** are part of the **flash.events** package, which does not need to be manually imported. Some other events, like **ToolTipEvent** and **DropDownEvent** are defined within **mx.events** and **spark.events** respectively. If you want to use one of these other event types in a function, you'll need to import the class first:

```
import spark.events.*;
import mx.events.*;
```

The ActionScript documentation shows the package each class is defined in.

Using specific types of event objects in your code will

- Provide additional functionality through added properties and methods
- Result in tighter, less bug-prone code
- Perform better

You'll see examples of this in action over the course of the rest of the chapter.

SYSTEM EVENTS

The past several pages have looked at how you handle events, but let's look at some of the specific events in more detail. As previously mentioned, there are two categories of events: system and user-generated. System events are not triggered by user behavior but rather by application occurrences. There are three events that are triggered by every component as they are created, in the following order:

- **preInitialize**
- **initialize**
- **creationComplete**

A **preInitialize** event is triggered when a component has just been created (because it has to exist in order to have events) but none of its children have been generated. The **initialize** event is triggered when a component has been created as have its immediate children, but the component hasn't been fully sized and drawn. In other words, the component has been fully realized save for being visually manifested. When this event is triggered, you can tweak some of the component's properties, like those used to size the component,

if need be. The **creationComplete** event occurs when a component has been created, as well as all of its direct descendents. Further, the component has been sized and fully drawn in the application.

Similar to how the event flow works, components are *initialized* from the outside in but *created* from the inside out. So if you have two **Button**s within a **VGroup** within the **Application**, the **Application** will be initialized, then the **VGroup**, then the **Button**s. Conversely, the **Button**s will be created (completely) first, then the **VGroup**, then the **Application** (**Figure 4.15**).

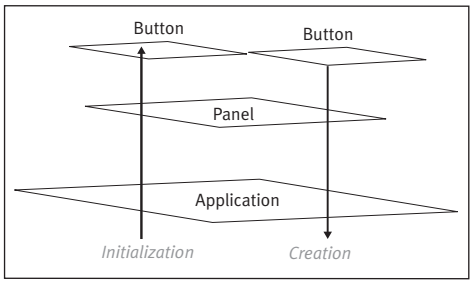

Figure 4.15

The application itself has a variation on **creationComplete**: **application Complete**. This event is triggered when every component in the application has been triggered. This event is frequently used, as programs will take this cue as the time to perform any kind of setup code:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    applicationComplete="init( )">
    <fx:Script>
    <![CDATA[
    private function init( ):void {
        // Do whatever.
    }
    ]]>
    </fx:Script>
```

Typically, a function named **init( )**—short for *initialize*, but that may be a bit confusing here—is called when the application is ready to run. It might set the values of variables, populate data-driven components, start network requests, etc.

tip

Another system event is ***show***, which is when a component switches from being invisible to being visible.

USER EVENTS

Already in this chapter and in Chapter 3, I've made frequent references to two user events: *click* and *change*. The former is a mouse event, occurring when the user clicks a component. The later is a general event type, and is triggered when the user changes something, like what's entered in a text box or selected in a drop-down menu. Let's look at user-generated events in more detail.

Keyboard and Mouse Events

Many of the user events are driven by keyboard and mouse activities (and "mouse" does include the range of cursor inputs, like trackpads, trackballs, and tablets). Here's a list of the most common keyboard and mouse-related events:

- **click**
- **doubleClick**
- **focusIn**
- **focusOut**
- **keyDown**
- **keyUp**
- **mouseDown**
- **mouseOut**
- **mouseOver**
- **mouseUp**
- **mouseWheel**
- **move**
- **rollOut**
- **rollover**
- **toolTipShow**
- **toolTipHide**

Each should be rather obvious. For some user actions several events come into play: pressing a key can trigger both **keyDown** and **keyUp**; clicking a component can entail **mouseOver**, **mouseDown**, **mouseUp**, **mouseOut**, and **click**.

As an example, let's create an application that when you mouseover a word, it displays the Spanish version of that word (**Figure 4.16**). The application will need to respond to two events: **mouseOver**, to show the Spanish word, and **mouseOut**, to show no word (**Figure 4.17**).

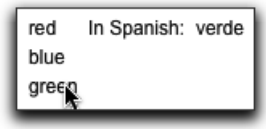

Figure 4.16

Figure 4.17

1. Create a new Flex project for the Web.

2. Within a **Script** block, define a function:

```
<fx:Script>
<![CDATA[
private function updateSpanish(spanishWord:String = ''):void {
    spanishWordLabel.text = spanishWord;
}
]]>
</fx:Script>
```

   The function expects one argument of type **String**. It has a default value of an empty string, making that argument optional. Within the function, the **text** property of the **spanishWordLabel** component will be assigned the received value.

3. Create an **HGroup** that contains a **VGroup** and two **Label**s:

```
<s:HGroup x="10" y="10">
    <s:VGroup mouseOut="updateSpanish()">
    </s:VGroup>
    <s:Label text="In Spanish: " /><s:Label id="spanishWordLabel" />
</s:HGroup>
```

   The **HGroup** is used to lay the elements out horizontally. Within it, there's a **VGroup**, which will contain the list of words, and two **Label**s. The first **Label** is a, um, label, for the second, which will actually display the Spanish version of the word. It's the only component that needs an **id** value.

   The **VGroup** has an event handler tied to the **mouseOut** event. When that event occurs on the **VGroup** or on any children of the **VGroup** (thanks to event bubbling), the **updateSpanish( )** function will be called. Since the function is called without passing along any values, the function will assign an empty string to the text property of the **spanishWordLabel** Label.

4. Within the **VGroup**, add three **Label**s:

```
<s:Label text="red" mouseOver="updateSpanish('rojo')" />
<s:Label text="blue" mouseOver="updateSpanish('azul')" />
<s:Label text="green" mouseOver="updateSpanish('verde')" />
```

tip

If you download the source code from the corresponding Web site (*www.dmcinsights.com/flex4*), you'll find this code in the **Ch04/Ch04_02** folder.

The three **Label**s differ in two ways: Each has a different text value and each sends a different string to the **updateSpanish( )** function, called when the cursor moves over the **Label**.

5. Save, compile, and run the application.

Keyboard and Mouse Event Objects

When you have keyboard and mouse events, you have the option of the event handler receiving a **KeyboardEvent** or **MouseEvent** object. Along with the properties inherited from the **Event** class, both keyboard and mouse objects have each of the following:

- **altKey**
- **ctrlKey**
- **shiftKey**

Each returns a Boolean value indicating if the key in question was also pressed down when the click, or other key press, occurred. For example, if you want to program your application so that Shift+clicking a component has a different effect, the event handler might be written like so:

```
private function handleClick(event:MouseEvent):void {
    if (event.shiftKey == true) {
        // Do this.
    } else {
        // Do this instead.
    }
}
```

As said earlier, mouse events also have **stageX** and **stageY** properties that store where, on the application, the click occurred, in pixels.

Keyboard events have their own **charCode** and **keyCode** properties. The latter is a numeric representation of which key on the keyboard was pressed. The former is a numeric representation of which character in the user's character set was pressed. In some cases, as with the capital letters, these numbers will be the same. For example, if you wanted to take some action when the user pressed Control+E (actually, the lowercase letter e, whose character code is 101), you could write the function like so:

```
private function handleKeys(event:KeyboardEvent):void {
    if ( (event.ctrlKey == true) && (event.charCode == 101) ) {
        // Do this.
    }
}
```

You can find the symbolic equivalents of the codes at *www.signar.se/blog/as-3-charcodes/*, among other places.

With keyboard-related events, you most likely want to set the handler on the entire application. For example, if pressing Control+T should do something, the whole program has to watch for that. That being said, be forewarned that if you're using the modifier keys and overriding default browser behavior, the results may not be expected. For example, you could assign the combination of Command+W (on Mac) or Control+W (on Windows) to have meaning within your Flash application. However, these are already used by the browser to close the window. In all likelihood, when the user presses that combination, the window will close and the Flash application will never get the chance to respond to the input.

Other User-Driven Events

There are a number of common events not specific to the keyboard or mouse:

- **copy**
- **cut**
- **paste**
- **select**
- **selectAll**
- **open**
- **close**
- **change**

These are all defined within the generic **Event** class, and therefore inherited by the other specific classes. The editing events—cut, copy, paste, and select all—are triggered when the user takes those actions, allowing the application to respond if need be. The open and close events apply to things like drop-down menus and combo boxes. And change has been demonstrated many times over by now.

There are four other specific event classes worth mentioning:

- **ColorPickerEvent**
- **DateChooserEvent**
- **DropDownEvent**

tip

In Chapter 10, "Creating Custom Code," you'll learn how to create your own event types.

A **ColorPickerEvent** object is created when the user makes a selection within a **ColorPicker** component. A **DateChooserEvent** object is created when the user makes a selection within a **DateChooser** component. And **DropDownEvent** (an update in Spark from **DropdownEvent** in Halo) applies to drop-down menus. All of these objects are manufactured when the component experiences a **change** event.

MANAGING EVENT HANDLERS WITH ACTIONSCRIPT

You can make the event-component-event handler connection in two ways. The first has been the focus thus far; create the association within the component's MXML:

```
<s:Label id="myLabel" x="20" y="120" click="moveMe(event)"
text="Click Me!" />
```

The alternative is to use ActionScript to make the association. Although the MXML approach is easy to understand, there are some good reasons to go the ActionScript route.

First of all, there are some occasions where you cannot identify event handlers in MXML. This would be the case if you were using ActionScript classes that aren't defined in MXML, or when you create objects on the fly.

Second, by using ActionScript, you can add *and remove* event handlers as needed. For example, you might have part of an application wherein the user selects one of three images by clicking it. Once the image has been selected, you could remove the event handlers from all of the images, so that the selection could not be changed.

A third benefit of using ActionScript in this way is that it further separates the application's behavior from its markup (i.e., its visual elements). In Web development, this concept is called *unobtrusive scripting* (there are other benefits to this approach in Web development that don't apply to Flash).

To add an event handler to a component using ActionScript, call the **addEventListener( )** method on the component. The method's first argument is the event to watch for, the second is the function to call when that event occurs:

```
someObject.addEventListener(EVENT_TYPE, someFunction);
```

You'll note by the capitalization that the event type is to be identified using a predefined constant. You'll find these constants listed under Public Constants in the ActionScript documentation; here are several examples:

- **Event.CHANGE**
- **KeyboardEvent.KEY_DOWN**
- **KeyboardEvent.KEY_UP**
- **MouseEvent.CLICK**
- **MouseEvent.DOUBLE_CLICK**
- **MouseEvent.MOUSE_DOWN**
- **MouseEvent.MOUSE_MOVE**
- **MouseEvent.MOUSE_OUT**
- **MouseEvent.MOUSE_OVER**
- **MouseEvent.MOUSE_UP**
- **MouseEvent.MOUSE_WHEEL**
- **MouseEvent.ROLL_OUT**
- **MouseEvent.ROLL_OVER**

The function to be called is the name of the function, without quotes, and without the parentheses (because it's a function *reference*, not a function *call*).

With this in mind, here is how you would replicate the previous bit of MXML, which associates the **move( )** function with the click event on the **myLabel Label**, using ActionScript:

```
myLabel.addEventListener(MouseEvent.CLICK, moveMe);
```

It's up to you to decide when it's appropriate to add event listeners, but you cannot do so before the component has been initialized. For that reason, you may want to establish the event handlers in an **init( )** function after the entire application has been created:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    applicationComplete="init( )">
    <fx:Script>
    <![CDATA[
```

code continues on the next page

tip

You can use a string to indicate the event type: '*click*', '*change*', *etc., but you're less likely to have bugs and problems if you use the constants.*

```
    private function init( ):void {
        myLabel.addEventListener(MouseEvent.CLICK, moveMe);
        // Add other event listeners.
    }
    ]]>
    </fx:Script>
```

Apply **addEventListener( )** multiple times to an object to add multiple event handlers:

```
someObj.addEventListener(MouseEvent.MOUSE_DOWN, doThis1);
someObj.addEventListener(MouseEvent.MOUSE_UP, doThis2);
```

Or the same function can be used for the same event on different components:

```
someObj1.addEventListener(MouseEvent.DOUBLE_CLICK, doThis);
someObj2.addEventListener(MouseEvent.DOUBLE_CLICK, doThis);
```

When you add event listeners in this manner, the event object will automatically be passed to the handling function. You need to write the function to accept that parameter. You can write it to accept either the generic **Event** or the specific event type, as appropriate:

```
private function moveMe(event:Event):void {
}
private function moveMe(event:MouseEvent):void {
}
```

Watching for Phases

The **addEventListener( )** method has two required arguments but three more optional ones. I want to look at the first of these. The next parameter is **useCapture**, which takes a Boolean value. If set to **true**, the listener will watch for events during the capture phase. If **false**, which is the default for most components, the listener will only watch for events during the target and bubble phases.

```
someObj.addEventListener(MouseEvent.CLICK, doThis, true);
```

This means that if you want to have a listener watch for events under all phases, you'll need to call **addEventListener( )** twice—once with a **false** value for **useCapture** (the default) and once with a **true** value:

```
someObj.addEventListener(MouseEvent.CLICK, doThis);
someObj.addEventListener(MouseEvent.CLICK, doThis, true);
```

Let's use this knowledge to recreate the event-reporting utility created earlier in the chapter.

1. Create a new Flex project for the Web.

2. In the **Application** tag, associate the **init( )** function with the complete creation of the application:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init( )">
```

   All of the event handlers are going to be assigned using ActionScript, which means that a function needs to be called once every component has been created.

3. Within a **Script** block, define the **init( )** function:

```
<fx:Script>
<![CDATA[
private function init( ):void {
    this.addEventListener(MouseEvent.CLICK, reportOnEvent);
    this.addEventListener(MouseEvent.CLICK, reportOnEvent, true);
    myVGroup.addEventListener(MouseEvent.CLICK, reportOnEvent);
    myVGroup.addEventListener(MouseEvent.CLICK, reportOnEvent, true);
    myLabel.addEventListener(MouseEvent.CLICK, reportOnEvent);
    myLabel.addEventListener(MouseEvent.CLICK, reportOnEvent, true);
    results.addEventListener(MouseEvent.CLICK, reportOnEvent);
    results.addEventListener(MouseEvent.CLICK, reportOnEvent, true);
}
]]>
</fx:Script>
```

   This function is called after the application has been created. Its job is to add event handlers to the application, **VGroup**, **Label**, and **RichText** components, just like in the earlier version of this program. For each element, **addEvent Listener( )** is being called twice: once with no value for **useCapture** (meaning it'll be **false**) and once with a **true** value. For every case, the **reportOnEvent( )** function is being associated with the mouse click.

   Note that the **Application** tag cannot be given an **id** value, but the special keyword **this** refers to the **Application**.

4. Still within the **Script** block, define the **reportOnEvent( )** function:

```
private function reportOnEvent(event:Event):void {
    results.text += "Target: " + event.target.id + "\ncurrentTarget: "
+ event.currentTarget.id + "\nPhase: " + event.eventPhase +
"\n----------\n";
}
```

   The function is defined exactly as it was before.

tip

If you download the source code from the corresponding Web site (*www.dmcinsights.com/flex4*), you'll find this code in the **Ch04/Ch04_03** folder.

tip

In OOP, the keyword ***this*** refers to the current object. In a Flex application, the current object is the application itself.

5. Define the components:

```
<s:VGroup id="myVGroup" paddingLeft="10" paddingTop="10">
    <s:Label id="myLabel" text="Click on Me!" />
    <s:Panel title="Results">
        <s:RichText id="results" />
    </s:Panel>
</s:VGroup>
```

These components are defined exactly as they were before, using the same **id** values, but they no longer have event listeners defined within the MXML.

6. Save, compile, and run the application.

7. Click the various components, and not any components at all, to see the results (**Figure 4.18**).

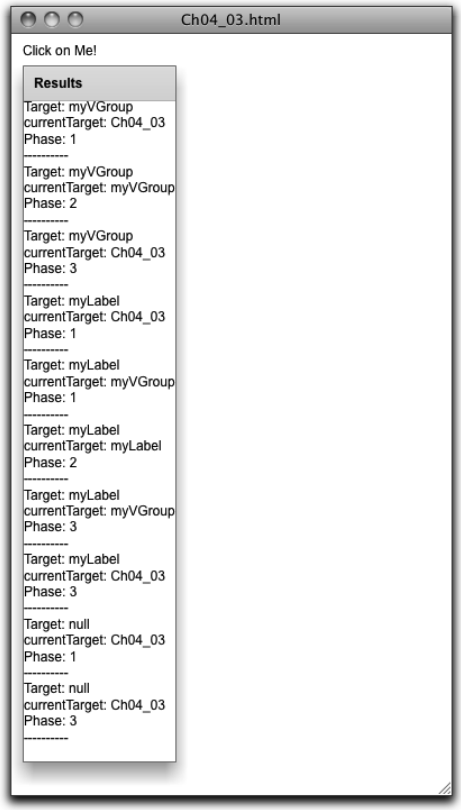

Figure 4.18

The figure shows the results after clicking the **VGroup** first (I clicked to the left of the **Panel**), the **Label** next, and, finally, outside of any component.

The **target** value for each is what you would expect. But now you have at least one capture-phase event (phase 1) for each click. And in the capture phase, just as in the bubble phase, the **currentTarget** value differs from the actual **target**.

EVENT PRIORITIES

The fourth argument to **addEventListener( )** is for assigning priorities to event handlers. If two events are triggered at the same time, like if you have click event handlers on both a **Button** and the **Panel** it's in, the listener with the higher priority will be called first. The value can be any integer, positive or negative, with the default value being 0.

Removing Event Handlers

To remove an event handler from an object, apply the **removeEventHandler( )** method, passing it the same two values as were used to add the event handler (three values if the **addEventListener( )** method):

```
someObj.removeEventListener(MouseEvent.CLICK, doThis);
someObj.removeEventListener(MouseEvent.CLICK, doThis, true);
```

To demonstrate how you might dynamically add and remove event handlers on the fly, this next application will either show, or not show, the answer to a series of (easy) math questions based upon whether a box is checked. If the box is checked, then event handlers are added so that when the user moves the cursor over a question, the answer is revealed (**Figure 4.19**). If the box is unchecked, the event handlers are removed so that moving the cursor over a question has no effect (**Figure 4.20**).

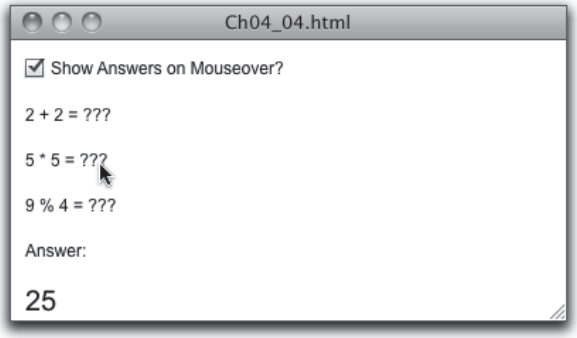

Figure 4.19

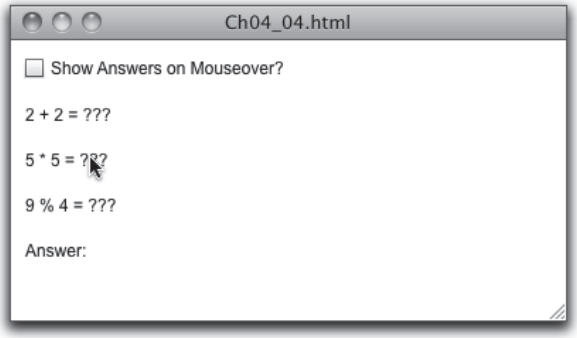

Figure 4.20

tip

The ***hasEventListener( )*** method returns a Boolean value indicating if the object has the indicated event listener. It takes the event type as its only argument.

tip

If you download the source code from the corresponding Web site (*www.dmcinsights.com/flex4*), you'll find this code in the **Ch04/Ch04_04** folder.

1. Create a new Flex project for the Web.

2. Define the components:

```
<s:VGroup x="10" y="10" gap="20">
    <s:CheckBox id="showAnswers" change="addRemoveHandlers( );"
    label="Show Answers on Mouseover?" />
    <s:Label id="question1" text="2 + 2 = ???" />
    <s:Label id="question2" text="5 * 5 = ???" />
    <s:Label id="question3" text="9 % 4 = ??? " />
    <s:Label text="Answer: " />
    <s:Label id="answerLabel" fontSize="20" />
</s:VGroup>
```

   This application has a handful of components, placed within a vertical group. Only one of these components—the check box—has an MXML-assigned event listener. When the check box is changed, the **addRemoveHandlers( )** function will be called.

   Each **Label** is given a unique **id** so that its answer can be displayed. The answer itself will be shown in the **answerLabel** component.

3. Within a **Script** block, define the **addRemoveHandlers( )** function:

```
<fx:Script>
<![CDATA[
private function addRemoveHandlers( ):void {
    if (showAnswers.selected == true) {
        question1.addEventListener(MouseEvent.MOUSE_OVER,
        showAnswer);
        question2.addEventListener(MouseEvent.MOUSE_OVER,
        showAnswer);
        question3.addEventListener(MouseEvent.MOUSE_OVER,
        showAnswer);
        question1.addEventListener(MouseEvent.MOUSE_OUT,
        clearAnswer);
        question2.addEventListener(MouseEvent.MOUSE_OUT,
        clearAnswer);
        question3.addEventListener(MouseEvent.MOUSE_OUT,
        clearAnswer);
    } else {
        question1.removeEventListener(MouseEvent.MOUSE_OVER,
        showAnswer);
```

```
            question2.removeEventListener(MouseEvent.MOUSE_OVER,
            showAnswer);
            question3.removeEventListener(MouseEvent.MOUSE_OVER,
            showAnswer);
            question1.removeEventListener(MouseEvent.MOUSE_OUT,
            clearAnswer);
            question2.removeEventListener(MouseEvent.MOUSE_OUT,
            clearAnswer);
            question3.removeEventListener(MouseEvent.MOUSE_OUT,
            clearAnswer);
        }
    }
    ]]>
    </fx:Script>
```

The function takes no arguments. Within the function, an **if-else** conditional either adds or removes event listeners based upon the **selected** property of the check box. If the box is selected, then two event listeners are added to each of the three components. If the box is not selected, then those same two event listeners are removed from each component.

The first event listener watches for a mouseover, at which point the **showAnswer( )** function will be called. The second event listener watches for a mouseout event, at which point the **clearAnswer( )** function will be called. This function removes the previously displayed answer.

tip

Through much more complicated code, you could streamline this process, but for demonstration purposes I'm going with a bit more code in the hopes that the process is clearer.

4. Also within the **Script** block, define the **showAnswer( )** function:

```
private function showAnswer(event:MouseEvent):void {
    var answer:String = '';
    switch (event.target.id) {
        case 'question1':
            answer = '4';
            break;
        case 'question2':
            answer = '25';
            break;
        case 'question3':
            answer = '1';
            break;
    }
    answerLabel.text = answer;
}
```

note

Even though the answers are all numbers, I create them as strings as they'll be assigned to a property that expects a string.

The goal of this function is to display an answer in **Label** by assigning a value to that component's **text** property. To determine what the right answer is, the function looks at the **event.target.id** property within a **switch**.

5. Still within the **Script** block, define the **clearAnswer( )** function:

```
private function clearAnswer(event:MouseEvent):void {
    answerLabel.text = '';
}
```

All this function does is assign an empty string to the **text** property of the **Label**. Even though it doesn't use the event object, the function will still receive one, as is always the case when using **addEventListener( )**.

6. Save, compile, and run the application.

CONTROLLING EVENTS

There's a lot more you can do with events than what I demonstrate in this chapter, although certainly this chapter's material covers the information you'd use the majority of the time. In Flash you can prevent event responses from taking place by calling the **preventDefault( )** method of an event object. This doesn't stop the event from happening, just stops whatever default behavior is associated with that event. To stop an event itself, you'd call the event object's **stopPropagation( )** or **stopImmediate Propagation( )** methods.

On the other side of the coin, you can dispatch events manually, if you need. You can even create your own event types, which you might do in conjunction with a custom component you've created.

INDEX

SYMBOLS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z