Essential ActionScript 3.0

Colin Moock

O'Reilly®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo
Adobe Developer Library, a copublishing partnership between O’Reilly Media Inc., and Adobe Systems, Inc., is the authoritative resource for developers using Adobe technologies. These comprehensive resources offer learning solutions to help developers create cutting-edge interactive web applications that can reach virtually anyone on any platform.

With top-quality books and innovative online resources covering the latest tools for rich-Internet application development, the Adobe Developer Library delivers expert training straight from the source. Topics include ActionScript, Adobe Flex®, Adobe Flash®, and Adobe Acrobat®.

Get the latest news about books, online resources, and more at http://adobedeveloperlibrary.com.
This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.
# Table of Contents

**Foreword** ................................................................. xv

**Preface** ................................................................. xix

**Part I.  ActionScript from the Ground Up**

1. **Core Concepts** ....................................................... 3
   - Tools for Writing ActionScript Code 3
   - Flash Client Runtime Environments 4
   - Compilation 5
   - Quick Review 6
   - Classes and Objects 6
   - Creating a Program 8
   - Packages 9
   - Defining a Class 11
   - Virtual Zoo Review 13
   - Constructor Methods 14
   - Creating Objects 16
   - Variables and Values 19
   - Constructor Parameters and Arguments 24
   - Expressions 26
   - Assigning One Variable’s Value to Another 28
   - An Instance Variable for Our Pet 30
   - Instance Methods 31
   - Members and Properties 42
   - Virtual Zoo Review 42
   - Break Time! 43
2. **Conditionals and Loops** ........................................ 44  
   | Conditionals .................................................. 44  
   | Loops .............................................................. 50  
   | Boolean Logic .................................................. 58  
   | Back to Classes and Objects ................................ 62  

3. **Instance Methods Revisited** .................................. 63  
   | Omitting the this Keyword .................................... 64  
   | Bound Methods .................................................. 66  
   | Using Methods to Examine and Modify an Object's State 68  
   | Get and Set Methods ............................................ 72  
   | Handling an Unknown Number of Parameters .............. 75  
   | Up Next: Class-Level Information and Behavior .......... 76  

4. **Static Variables and Static Methods** ....................... 77  
   | Static Variables ................................................ 77  
   | Constants .......................................................... 80  
   | Static Methods ................................................... 82  
   | Class Objects ..................................................... 85  
   | C++ and Java Terminology Comparison ...................... 86  
   | On to Functions ................................................... 86  

5. **Functions** .......................................................... 87  
   | Package-Level Functions ........................................ 88  
   | Nested Functions ................................................ 90  
   | Source-File-Level Functions ................................... 91  
   | Accessing Definitions from Within a Function ............ 92  
   | Functions as Values ............................................. 93  
   | Function Literal Syntax ....................................... 93  
   | Recursive Functions ............................................. 95  
   | Using Functions in the Virtual Zoo Program ............... 96  
   | Back to Classes .................................................. 100  

6. **Inheritance** ..................................................... 101  
   | A Primer on Inheritance ....................................... 101  
   | Overriding Instance Methods ................................ 105  
   | Constructor Methods in Subclasses ......................... 108  
   | Preventing Classes from Being Extended and Methods from Being Overridden 112
Table of Contents

Subclassing Built-in Classes 113
The Theory of Inheritance 114
Abstract Not Supported 120
Using Inheritance in the Virtual Zoo Program 121
Virtual Zoo Program Code 126
It’s Runtime! 129

7. Compiling and Running a Program 130
Comiling with the Flash Authoring Tool 130
Compiling with Flex Builder 2 131
Compiling with mxmlc 133
Compiler Restrictions 134
The Compilation Process and the Classpath 134
Strict-Mode Versus Standard-Mode Compilation 135
The Fun’s Not Over 136

8. Datatypes and Type Checking 137
Datatypes and Type Annotations 138
Untyped Variables, Parameters, Return Values, and Expressions 142
Strict Mode’s Three Special Cases 143
Warnings for Missing Type Annotations 144
Detecting Reference Errors at Compile Time 145
Casting 146
Conversion to Primitive Types 150
Default Variable Values 153
null and undefined 153
Datatypes in the Virtual Zoo 154
More Datatype Study Coming Up 158

9. Interfaces 159
The Case for Interfaces 159
Interfaces and Multidatatype Classes 161
Interface Syntax and Use 162
Another Multiple-Type Example 165
More Essentials Coming 171

10. Statements and Operators 172
Statements 172
Operators 174
Up Next: Managing Lists of Information 185
# Table of Contents

## 11. Arrays 186
- What Is an Array? 186
- The Anatomy of an Array 187
- Creating Arrays 187
- Referencing Array Elements 189
- Determining the Size of an Array 191
- Adding Elements to an Array 193
- Removing Elements from an Array 197
- Checking the Contents of an Array with the `toString()` Method 199
- Multidimensional Arrays 200
- On to Events 201

## 12. Events and Event Handling 202
- ActionScript Event Basics 202
- Accessing the Target Object 209
- Accessing the Object That Registered the Listener 212
- Preventing Default Event Behavior 213
- Event Listener Priority 214
- Event Listeners and Memory Management 216
- Custom Events 221
- Type Weakness in ActionScript’s Event Architecture 233
- Handling Events Across Security Boundaries 236
- What’s Next? 240

## 13. Exceptions and Error Handling 241
- The Exception-Handling Cycle 241
- Handling Multiple Types of Exceptions 244
- Exception Bubbling 253
- The finally Block 258
- Nested Exceptions 260
- Control-Flow Changes in `try/catch/finally` 264
- Handling a Built-in Exception 267
- More Gritty Work Ahead 268

## 14. Garbage Collection 269
- Eligibility for Garbage Collection 269
- Incremental Mark and Sweep 272
- Disposing of Objects Intentionally 273
- Deactivating Objects 274
15. Dynamic ActionScript ................................................. 279
   Dynamic Instance Variables 280
   Dynamically Adding New Behavior to an Instance 284
   Dynamic References to Variables and Methods 286
   Using Dynamic Instance Variables to Create Lookup Tables 287
   Using Functions to Create Objects 289
   Using Prototype Objects to Augment Classes 291
   The Prototype Chain 292
   Onward! 294

16. Scope ........................................................................... 295
   Global Scope 296
   Class Scope 297
   Static Method Scope 298
   Instance Method Scope 298
   Function Scope 299
   Scope Summary 300
   The Internal Details 300
   Expanding the Scope Chain via the with Statement 302
   On to Namespaces 303

17. Namespaces ............................................................... 304
   Namespace Vocabulary 304
   ActionScript Namespaces 305
   Creating Namespaces 307
   Using a Namespace to Qualify Variable and Method Definitions 310
   Qualified Identifiers 312
   A Functional Namespace Example 314
   Namespace Accessibility 317
   Qualified-Identifier Visibility 321
   Comparing Qualified Identifiers 322
   Assigning and Passing Namespace Values 323
   Open Namespaces and the use namespace Directive 334
   Namespaces for Access-Control Modifiers 338
   Applied Namespace Examples 341
   Final Core Topics 352
18. XML and E4X ................................................................. 353
   Understanding XML Data as a Hierarchy 353
   Representing XML Data in E4X 355
   Creating XML Data with E4X 357
   Accessing XML Data 359
   Processing XML with for-each-in and for-in 377
   Accessing Descendants 379
   Filtering XML Data 383
   Traversing XML Trees 386
   Changing or Creating New XML Content 387
   Loading XML Data 397
   Working with XML Namespaces 398
   Converting XML and XMLList to a String 404
   Determining Equality in E4X 407
   More to Learn 410

19. Flash Player Security Restrictions ................................. 411
   What’s Not in This Chapter 412
   The Local Realm, the Remote Realm, and Remote Regions 412
   Security-Sandbox-Types 413
   Security Generalizations Considered Harmful 415
   Restrictions on Loading Content, Accessing Content as Data, Cross-Scripting, and Loading Data 416
   Socket Security 422
   Example Security Scenarios 422
   Choosing a Local Security-Sandbox-Type 425
   Distributor Permissions (Policy Files) 429
   Creator Permissions (allowDomain()) 444
   Import Loading 446
   Handling Security Violations 448
   Security Domains 450
   Two Common Security-Related Development Issues 452
   On to Part II! 454
Part II. Display and Interactivity

20. The Display API and the Display List ................................. 457
   Display API Overview 458
   The Display List 462
   Containment Events 487
   Custom Graphical Classes 499
   Go with the Event Flow 501

21. Events and Display Hierarchies ......................................... 502
   Hierarchical Event Dispatch 502
   Event Dispatch Phases 503
   Event Listeners and the Event Flow 505
   Using the Event Flow to Centralize Code 511
   Determining the Current Event Phase 514
   Distinguishing Events Targeted at an Object from Events Targeted at That
   Object’s Descendants 516
   Stopping an Event Dispatch 518
   Event Priority and the Event Flow 522
   Display-Hierarchy Mutation and the Event Flow 523
   Custom Events and the Event Flow 526
   On to Input Events 530

22. Interactivity ............................................................... 531
   Mouse-Input Events 532
   Focus Events 548
   Keyboard-Input Events 555
   Text-Input Events 565
   Flash Player-Level Input Events 580
   From the Program to the Screen 586

23. Screen Updates ............................................................. 587
   Scheduled Screen Updates 587
   Post-Event Screen Updates 596
   Redraw Region 600
   Optimization with the Event.RENDER Event 601
   Let’s Make It Move! 609
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.</td>
<td>Programmatic Animation</td>
<td>610 - 628</td>
</tr>
<tr>
<td>25.</td>
<td>Drawing with Vectors</td>
<td>629 - 647</td>
</tr>
<tr>
<td>26.</td>
<td>Bitmap Programming</td>
<td>648 - 694</td>
</tr>
<tr>
<td>27.</td>
<td>Text Display and Input</td>
<td>695 - 793</td>
</tr>
</tbody>
</table>
## 28. **Loading External Display Assets** .......................... 762
   - Using Loader to Load Display Assets at Runtime 763
   - Compile-Time Type-Checking for Runtime-Loaded Assets 781
   - Accessing Assets in Multiframe .swf Files 790
   - Instantiating a Runtime-Loaded Asset 793
   - Using Socket to Load Display Assets at Runtime 796
   - Removing Runtime Loaded .swf Assets 806
   - Embedding Display Assets at CompileTime 807
   - On to Part III 818

## Part III. **Applied ActionScript Topics**

## 29. **ActionScript and the Flash Authoring Tool** .................. 821
   - The Flash Document 821
   - Timelines and Frames 822
   - Timeline Scripting 826
   - The Document Class 828
   - Symbols and Instances 832
   - Linked Classes for Movie Clip Symbols 834
   - Accessing Manually Created Symbol Instances 838
   - Accessing Manually Created Text 844
   - Programmatic Timeline Control 845
   - Instantiating Flash Authoring Symbols via ActionScript 847
   - Instance Names for Programmatically Created Display Objects 848
   - Linking Multiple Symbols to a Single Superclass 849
   - The Composition-Based Alternative to Linked Classes 851
   - Preloading Classes 852
   - Up Next: Using the Flex Framework 855
30. A Minimal MXML Application .......................................... 856
   The General Approach ............................................. 856
   A Real UI Component Example ..................................... 859
   Sharing with Your Friends ......................................... 860

31. Distributing a Class Library ........................................ 861
   Sharing Class Source Files ........................................ 862
   Distributing a Class Library as a .swc File .................... 863
   Distributing a Class Library as a .swf File .................... 867
   But Is It Really Over? ............................................... 873

Appendix ........................................................................ 875

Index ........................................................................... 891
CHAPTER 21

Events and Display Hierarchies

In Chapter 12, we studied ActionScript’s built-in event architecture in general terms. In this chapter, we’ll take a closer look at how that event architecture specifically caters to objects in display hierarchies.

ActionScript’s system of dispatching events through an object hierarchy, as described in this chapter, is based on the W3C Document Object Model (DOM) Level 3 Events Specification, available at http://www.w3.org/TR/DOM-Level-3-Events.

Hierarchical Event Dispatch

As we saw in Chapter 12, when ActionScript dispatches an event targeted at an object that is not part of a display hierarchy, that target is the sole object notified of the event. For example, when a Sound object’s sound finishes playing, ActionScript dispatches an Event.COMPLETE event targeted at the associated SoundChannel object. The SoundChannel object is not part of a display hierarchy, so it is the sole object notified that the event occurred.

By contrast, when ActionScript dispatches an event targeted at an object that is part of a display hierarchy, that target and all of its display hierarchy ancestors are notified that the event occurred. For example, if a Sprite object contains a TextField object, and the user clicks the TextField object, both the TextField object (the event target) and the Sprite object (the event target’s ancestor) are notified that the mouse click occurred.

ActionScript’s hierarchical event dispatch system enables every display object container to register event listeners that handle events targeted at its descendant display objects. For example, a Sprite representing a dialog box might register a listener that handles mouse click events targeted at a nested “OK button” control. Or a Sprite representing a login form might register a listener that handles focus events targeted at nested input fields. This centralized architecture helps reduce code repetition, par-
particularly for code that responds to user input events. In the later section “Using the Event Flow to Centralize Code,” we’ll study a code example that demonstrates the benefits of centralized event handling. But first, let’s cover the basics of hierarchical event dispatch and registration.

In this chapter, the terms “ancestor” and “descendant” refer primarily to objects in a display hierarchy, not to superclasses and subclasses in an inheritance hierarchy. To avoid confusion, this chapter sometimes uses the informal terms “display ancestor” and “display descendant” when referring to ancestor objects and descendant objects in a display hierarchy.

**Event Dispatch Phases**

As we just learned, when ActionScript dispatches an event targeted at an object in a display hierarchy, it notifies not just that target but also its display ancestors. The process by which the target and its ancestors are notified of the event is broken into three distinct phases. In the first phase of the event dispatch, known as the *capture phase*, each of the target’s ancestors is notified that the event has occurred. Once the target object’s ancestors have all been notified of the event, then the second phase of the event dispatch, known as the *target phase*, begins. During the target phase, ActionScript notifies the target object that the event occurred.

For some event types, the event dispatch ends once the target phase is complete. For other event types, the event dispatch continues into a third phase, known as the *bubbling phase*. During the bubbling phase, the ancestors of the target object are notified that the target successfully received the event notification. Events with a bubbling phase are known as *bubbling events*; events without a bubbling phase are known as *nonbubbling events*.

The four event types `Event.ACTIVATE`, `Event.DEACTIVATE`, `Event.ENTER_FRAME`, and `Event.RENDER`, have a target phase only. All other event dispatches targeted at an object in a display hierarchy have a capture phase and a target phase. Some event types also have a bubbling phase.

The order in which objects are notified of an event during an event dispatch is governed by the event phase. During the capture phase, ancestors are notified in an order that starts from the root of the target’s display hierarchy and proceeds down through each descendant to the target’s direct parent. During the target phase, the target, itself is notified. During the bubbling phase, ancestors are notified in the opposite order of the capture phase, from the target’s direct parent up to the root of the hierarchy. The process by which event notification passes down through a target’s ancestors (capture phase), to the target (target phase), and then back up through the target’s ancestors (bubbling phase) is known as the *event flow*. As event notification passes through event flow, the event is said to *propagate* from object to object.
Let's consider a simple event-flow example. Suppose the Stage instance contains a Sprite object that contains a TextField object, as depicted in Figure 21-1. As Figure 21-1 shows, the root of the TextField object’s display hierarchy is the Stage instance, and the TextField’s direct parent is the Sprite object.

Figure 21-1. A sample display hierarchy

Now further suppose that the user enters some text into the TextField, causing the Flash runtime to dispatch a TextEvent.TEXT_INPUT event targeted at the TextField object. Because the TextField object is part of a display hierarchy, the event passes through the event flow. During the first phase of the event dispatch (the capture phase), the Stage instance, then the Sprite instance are notified of the event. During the second phase of the dispatch (the target phase), the TextField itself is notified of the event. Finally, during the third phase of the dispatch (the bubbling phase), the Sprite instance, then the Stage instance are notified that the target received event notification. In all, five event notifications are carried out during the TextEvent.TEXT_INPUT event dispatch, as depicted in Figure 21-2.

Figure 21-2. Event flow for the TextEvent.TEXT_INPUT event
Event Listeners and the Event Flow

As we’ve just seen, during an event dispatch targeted at a given display object, that object’s display ancestors receive event notification during the capture phase and potentially also during the bubbling phase (if the event is a bubbling event). Accordingly, when we register a listener with an event target’s ancestor we must indicate whether that listener should be triggered during the capture phase or the bubbling phase.

To register a listener with an event target’s ancestor for the capture phase of an event dispatch, we set `addEventListener()`’s third parameter, `useCapture`, to `true`, as in:

```javascript
theAncestor.addEventListener(theEvent, theListener, true)
```

The preceding line of code causes `theListener()` to be executed whenever ActionScript dispatches `theEvent` targeted at one of `theAncestor`’s descendants, before that descendant receives notification of the event.

To register a listener with an event target’s ancestor for the bubbling phase of an event dispatch, we set `addEventListener()`’s third parameter to `false`, as in:

```javascript
theAncestor.addEventListener(theEvent, theListener, false)
```

Alternatively, because `useCapture`’s default value is `false`, we can simply omit the `useCapture` argument, as in:

```javascript
theAncestor.addEventListener(theEvent, theListener)
```

The preceding line of code causes `theListener()` to be executed whenever ActionScript dispatches `theEvent` targeted at one of `theAncestor`’s descendants, after that descendant receives notification of the event.

For brevity over the remainder of this chapter, we’ll use the unofficial term `ancestor listener` to mean “an event listener registered with an event target’s display ancestor.” Likewise, we’ll use the term `target listener` to mean “an event listener registered directly with an event target.”

When registering an ancestor listener for a `nonbubbling` event, we always register for the capture phase (i.e., set `useCapture` to `true`); otherwise, the listener will not be triggered. When registering an ancestor listener for a `bubbling` event, we choose either capture-phase notification (`useCapture` set to `true`) or bubbling-phase notification (`useCapture` set to `false`), or both, according to the needs of the application.

The capture phase gives ancestor listeners a chance to process an event before the event target’s listeners have responded to it. Typically, capture phase listeners are used to conditionally stop an event from ever reaching its target. For example, a panel widget with an “enabled” state and a “disabled” state might use a capture-phase listener to prevent the panel’s descendants from receiving mouse events when the panel is disabled. (We’ll learn how to stop events in the later section “Stopping an Event Dispatch.”)
By contrast, the bubbling phase gives ancestor listeners a chance to process an event after the event target’s listeners have responded to it. Typically, the bubbling phase is used to respond to changes in the state of the target object before the program continues, and before the screen is updated. For example, a panel widget containing draggable icons might use a bubbling-phase listener to trigger automatic icon-alignment after a specific icon has been dragged.

Unlike ancestor listeners, listeners registered with an event target can be triggered during a single phase only—the target phase. To register a listener with an event target for the target phase of an event dispatch, we register that listener using `addEventListener()` with `useCapture` set to `false`—exactly as if we were registering an ancestor listener for notification during the bubbling phase. The following generalized code shows the approach:

```
theEventTarget.addEventListener(theEvent, theListener, false)
```

Or, simply:

```
theEventTarget.addEventListener(theEvent, theListener)
```

The preceding line of code causes `theListener()` to be executed whenever ActionScript dispatches `theEvent` targeted at `theEventTarget`, after `theEventTarget`’s ancestors receive capture-phase notification of the event.

The following sections present a variety of `useCapture` examples and discuss several phase-specific event-registration topics.

**Registering an Ancestor Listener for the Capture Phase**

As we’ve just learned, to register a given ancestor listener for capture-phase event notification, we set `addEventListener()`’s `useCapture` parameter to `true`, as in:

```
theAncestor.addEventListener(theEvent, theListener, true)
```

Now let’s apply that code to a working example. For a sample display hierarchy, we’ll use the scenario depicted earlier in Figure 21-1, where the `Stage` instance contains a `Sprite` object that contains a `TextField` object. Example 21-1 shows the code we use to create the hierarchy.

**Example 21-1. A sample display hierarchy**

```actionscript
// Create the Sprite
var theSprite:Sprite = new Sprite();

// Create the TextField
var theTextField:TextField = new TextField();
```

506  |  Chapter 21: Events and Display Hierarchies
Now suppose we want to register a function, `textInputListener()`, with `theSprite` for TextEvent.TEXT_INPUT events. Here’s the `textInputListener()` function:

```actionscript
private function textInputListener (e:TextEvent):void {
    trace("The user entered some text");
}
```

We want `textInputListener()` to be triggered during the capture phase (i.e., before the TextField is notified), so we use the following code to register it:

```actionscript
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, true)
```

The preceding line of code causes `textInputListener()` to be executed whenever ActionScript dispatches a TextEvent.TEXT_INPUT event targeted at the TextField, before the TextField receives notification of the event.

### Registering an Ancestor Listener for the Bubbling Phase

Recall that to register a given ancestor listener for bubbling-phase event notification, we set `addEventListener()`’s `useCapture` parameter to false, as in:

```actionscript
theAncestor.addEventListener(theEvent, theListener, false)
```

Continuing again with our TextField from Example 21-1, suppose we want to register `textInputListener()` with theSprite for TextEvent.TEXT_INPUT events, and we want `textInputListener()` to be triggered during the bubbling phase (i.e., after the TextField is notified). We use the following code:

```actionscript
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, false)
```

Or, we could do the same thing by omitting the `useCapture` parameter value entirely:

```actionscript
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener)
```

The preceding line of code causes `textInputListener()` to be executed whenever ActionScript dispatches a TextEvent.TEXT_INPUT event targeted at the TextField, but after theTextField receives notification of the event.

Note that if the TextEvent.TEXT_INPUT event were a nonbubbling event, then `textInputListener()` would never be triggered. It’s worth repeating what we learned earlier: if an ancestor listener registers for a nonbubbling event with `useCapture` either
omitted or set to false, then that listener will never be triggered. In order for an ancestor listener to be triggered when a nonbubbling event is dispatched, it must register for the capture phase by setting useCapture to true.

To determine whether an event is a bubbling or nonbubbling event, we can either:

• Consult the event’s entry in Adobe’s ActionScript Language Reference.
• Handle the event with an event listener during either the capture or target phases, and check the bubbles variable of the Event object passed to the listener.

If bubbles is true, then the event bubbles; otherwise, the event does not bubble.

The following code shows the latter technique:

```actionscript
// Register a function, clickListener(), with the Stage instance for
// MouseEvent.CLICK events. Note that someDisplayObject must be on the
// display list in order to access the Stage instance.
someDisplayObject.stage.addEventListener(MouseEvent.CLICK, clickListener);

// ...later in the code, define clickListener()
private function clickListener (e:MouseEvent):void {
  // When the event occurs, check if it is a bubbling event
  if (e.bubbles) {
    trace("The MouseEvent.CLICK event is a bubbling event.");
  } else {
    trace("The MouseEvent.CLICK event is a non-bubbling event.");
  }
}
```

For convenient reference, Adobe’s ActionScript Language Reference lists the value of the Event class’s instance variable bubbles under all built-in event entries. As a general rule, most built-in events targeted at display objects bubble.

### Registering an Ancestor Listener for Both the Capture Phase and the Bubbling Phase

To specify that an ancestor listener should be triggered both during the capture phase and the bubbling phase (i.e., before and after the target receives the event notification), we must register that listener twice—once with useCapture set to true and once with useCapture set to false. For example, returning to our TextField scenario, suppose we want to register our textInputListener() listener with theSprite for TextEvent.TEXT_INPUT events, and we want textInputListener() to be triggered during both the capture phase and the bubbling phase. We use the following code:

```actionscript
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, true)
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, false)
```

If an ancestor listener wishes to be triggered during both the capture phase and the bubbling phase of an event dispatch, it must register for the event twice.
Registering a Listener with the Event Target

Recall that to register a target listener for target-phase notification, we set `addEventListener()`'s `useCapture` parameter to false, as in:

```javascript
theEventTarget.addEventListener(theEvent, theListener, false)
```

Hence, in our ongoing `TextField` scenario, to register `textInputListener()` with `theTextField` for `TextEvent.TEXT_INPUT` events, we use the following code:

```javascript
theTextField.addEventListener(TextEvent.TEXT_INPUT, textInputListener, false)
```

or, simply:

```javascript
theTextField.addEventListener(TextEvent.TEXT_INPUT, textInputListener)
```

The preceding code causes `textInputListener()` to be executed whenever ActionScript dispatches a `TextEvent.TEXT_INPUT` event targeted at `theTextField`. The `textInputListener()` method executes after the `Stage` instance and `theSprite` have received capture-phase notification, but before the `Stage` instance and `theSprite` receive bubbling-phase notification.

The Dual Purpose of the `useCapture` Parameter

As shown in the preceding two sections, `addEventListener()`'s `useCapture` parameter is set to `false` in two different circumstances:

- When registering an ancestor listener to be triggered during the bubbling phase
- When registering a target listener to be triggered during the target phase

Therefore, when a listener registers for an event with `useCapture` set to `false`, that listener will be triggered when the event is dispatched, and either of the following is true:

- The event target is the object with which the listener registered (in this case, the listener is triggered during the target phase).
- The event target is a descendant of the object with which the listener registered (in this case, the listener is triggered during the bubbling phase, after that descendant has processed the event).

For example, the following code registers `clickListener()` with the `Stage` instance for `MouseEvent.CLICK` events:

```javascript
someDisplayObject.stage.addEventListener(MouseEvent.CLICK, clickListener, false);
```
Because `useCapture` is `false`, `clickListener()` will be triggered in both of the following situations:

- When the user clicks the display area (in which case the Flash runtime dispatches an event targeted at the `Stage` instance).
- When the user clicks any on-screen display object (in which case the Flash runtime dispatches an event targeted at the clicked object, which is always a descendant of the `Stage` instance).

Notice that although `clickListener()` registered with a single object (the `Stage` instance), at runtime `clickListener()` might be triggered by event dispatches targeted at that object or at that object’s descendants! Therefore, in some cases, a listener function must include code to ignore event dispatches in which it has no interest. We’ll study the code required to ignore event dispatches in the later section “Distinguishing Events Targeted at an Object from Events Targeted at That Object’s Descendants.”

### Removing Event Listeners

When unregistering an event listener from an object in a display hierarchy, we must indicate whether that listener originally registered to be triggered during the capture phase or during the target or bubbling phases. To do so, we use `removeEventListener()`’s third parameter, `useCapture`, which mirrors `addEventListener()`’s `useCapture` parameter.

If the listener being unregistered was originally registered for the capture phase (i.e., with `addEventListener()`’s `useCapture` parameter set to `true`), we must unregister it with `removeEventListener()`’s `useCapture` parameter set to `true`. If the listener was originally registered to be triggered during the target or bubbling phases (i.e., with `addEventListener()`’s `useCapture` parameter set to `false`), we must unregister it with `removeEventListener()`’s `useCapture` parameter set to `false`.

> When unregistering a listener, always set `removeEventListener()`’s `useCapture` parameter to match the value used for `useCapture` when `addEventListener()` was originally invoked.

For example, in the following code, we register `clickListener()` with `someDisplayObject` for the capture phase by specifying `true` as the value of `addEventListener()`’s `useCapture` parameter:

```javascript
someDisplayObject.addEventListener(MouseEvent.CLICK, clickListener, true);
```

Accordingly, when unregistering `clickListener()` from `someDisplayObject`, we must specify `true` as the value of `removeEventListener()`’s `useCapture` parameter:
someDisplayObject.removeEventListener(MouseEvent.CLICK,
clickListener,
true);

When unregistering an event listener that has registered twice with the same object (to be triggered during both the capture phase and the target or bubbling phases), then we must, likewise, invoke removeEventListener() twice. For example, the following code registers a MouseEvent.CLICK listener twice with the Stage instance, to be triggered during both the capture phase and the target or bubbling phases:

someDisplayObject.stage.addEventListener(MouseEvent.CLICK,
clickListener,
true);
someDisplayObject.stage.addEventListener(MouseEvent.CLICK,
clickListener,
false);

The following code removes both preceding MouseEvent.CLICK event listeners. Because clickListener() was registered separately for both the capture phase and the target or bubbling phases, it must also unregister separately for those phases.

someDisplayObject.stage.removeEventListener(MouseEvent.CLICK,
clickListener,
true);
someDisplayObject.stage.removeEventListener(MouseEvent.CLICK,
clickListener,
false);

Each listener registration performed with addEventListener() is treated separately, requiring its own removeEventListener() invocation for unregistration.

Now that we're familiar with the basics of the event flow, let's look at an example showing how it can help centralize code in a real-world application.

Using the Event Flow to Centralize Code

If you're waiting for room in a fully booked hotel, it's easier to ask the hotel manager to tell you when a vacancy opens up than it is to ask every guest in the hotel to tell you when they leave. Likewise, when handling event dispatches, it's often more efficient to register event listeners with a display object container than it is to register with each of its descendants.

For example, suppose we're building a simple checkbox control, comprised of the following two classes:

- CheckBox, a Sprite subclass that acts as a container for the entire control
- CheckBoxIcon, a Sprite subclass that represents the checkbox's graphical icon
At runtime, each CheckBox instance creates two child objects: a CheckBoxIcon instance for the checkbox’s icon and a TextField instance for the checkbox’s text label. For reference, let’s call the main CheckBox instance container and its two children icon and label. Figure 21-3 shows our checkbox control.

![Figure 21-3. Objects in the Checkbox control](image)

We want our checkbox to be easy to use, so we design it to toggle on or off when the user clicks either the checkbox icon or the checkbox label. Accordingly, in our implementation, we must detect mouse-click events targeted at both icon and label. To do so, we could register a separate mouse-click listener with each of those objects. However, registering two event listeners would increase development time and, due to the repetition of near-identical event registration code, increase the potential for bugs in our checkbox. To avoid repetitive code, we can instead handle all mouse-click events through a single listener registered with container. Because container is a display ancestor of both icon and label, it is informed any time the Flash runtime dispatches a mouse-click event targeted at either of those objects. Whenever container’s mouse-click listener runs, we know that either the icon or the label was clicked, and we can toggle the checkbox on or off in response.

Example 21-2 shows the code for our example checkbox, with event-handling sections in bold.

Example 21-2. Handling a Checkbox’s events hierarchically

```java
// File CheckBox.as
package {
import flash.display.*;
import flash.events.*;
import flash.text.*;

// A very simple checkbox widget
public class CheckBox extends Sprite {
    private var label:TextField; // The checkbox’s text label
    private var icon:CheckBoxIcon; // The checkbox’s graphical icon
    private var checked:Boolean; // Flag indicating whether the
                                  // checkbox is currently checked

    // Constructor
    public function CheckBox (msg:String) {
        // When first created, the checkbox is not checked
        checked = false;

        // Create the graphical icon
        icon = new CheckBoxIcon();
    }
}
```
Example 21-2. Handling a Checkbox's events hierarchically (continued)

    // Create the text label
    label = new TextField();
    label.text = msg;
    label.autoSize = TextFieldAutoSize.LEFT;
    label.selectable = false;

    // Position the text label next to the graphical icon
    label.x = icon.x + icon.width + 5;

    // Add the label and icon to this object as display children
    addChild(icon);
    addChild(label);

    // Start listening for mouse click event dispatches targeted at this
    // object or any of its children (i.e., the label or the icon)
    addEventListener(MouseEvent.CLICK, clickListener);
}

    // Handles mouse click events. This method runs whenever either the
    // label or the icon is clicked.
    private function clickListener (e:MouseEvent):void {
        if (checked) {
            icon.uncheck();
            checked = false;
        } else {
            icon.check();
            checked = true;
        }
    }
}

// File CheckBoxIcon.as
package {
    import flash.display.*;

    // The graphical icon for a checkbox widget
    public class CheckBoxIcon extends Sprite {

        // Constructor
        public function CheckBoxIcon () {
            uncheck();
        }

        // Draws a checkbox icon in the "checked" state
        public function check ():void {
            graphics.clear();
            graphics.lineStyle(1);
            graphics.beginFill(0xFFFFFF);
            graphics.drawFill(0x000000);
            graphics.drawRect(0, 0, 15, 15);
            graphics.endFill();
        }
    }
}
We've now covered the basics of ActionScript's hierarchical-event-dispatch system, but there are several more topics left to explore. Let's soldier on.

### Determining the Current Event Phase

As we learned in the earlier section “Registering an Ancestor Listener for Both the Capture Phase and the Bubbling Phase,” by invoking `addEventListener()` twice, we can register a single event-listener function to be executed during both the capture phase and bubbling phases of an event dispatch. Similarly, in the section “The Dual Purpose of the useCapture Parameter,” we learned that when an event listener is registered with `useCapture` set to `false`, that listener might be triggered during the target phase or during the bubbling phase of an event dispatch. Hence, when an event-listener function is executed in response to an event, the current event phase is not always known. Accordingly, ActionScript provides the `Event` class’s instance variable `eventPhase`, which can be used within an event listener function to deduce the current event phase.

The `eventPhase` variable indicates whether the current event dispatch is in the capture phase, the target phase, or the bubbling phase. When the event dispatch is in the capture phase, `eventPhase` is set to `EventPhase.CAPTURING_PHASE`, indicating that...
the target object has not yet received event notification. When the event dispatch is in the target phase, `eventPhase` is set to `EventPhase.AT_TARGET`, indicating that the target object is currently processing the event. When the event dispatch is in the bubbling phase, `eventPhase` is set to `EventPhase.BUBBLING_PHASE`, indicating that the target object has finished processing the event.

Typically, the `EventPhase.CAPTURING_PHASE`, `EventPhase.AT_TARGET`, and `EventPhase.BUBBLING_PHASE` constants have the integer values 1, 2, and 3, respectively, but those values are considered subject to change, and should never be used directly. Instead, to determine the current event phase within an event listener function, always compare the `eventPhase` variable to the `EventPhase` class constants. For example, always use code like this:

``` ActionScript 
private function someListener (e:Event):void {
    if (e.eventPhase == EventPhase.AT_TARGET) {
        // This listener was triggered during the target phase...
    }
}
```

And never use code like this:

``` ActionScript 
private function someListener (e:Event):void {
    // Bad code! Never use the EventPhase constant values directly!
    if (e.eventPhase == 2) {
        // This listener was triggered during the target phase...
    }
}
```

The following code demonstrates the general use of the `eventPhase` variable. First, the code adds a `TextField` object to the `Stage` instance. Then the code registers `clickListener()` with the `Stage` instance for capture-phase `MouseEvent.CLICK` event notification. Finally, the code registers `clickListener()` with the `Stage` instance for target-phase and bubbling-phase `MouseEvent.CLICK` event notification.

When `clickListener()` executes, it outputs the current phase. Notice that the current phase is determined by comparing `eventPhase` to the three `EventPhase` class constants.

``` ActionScript 
var t:TextField = new TextField();
t.text = "click here";
t.autoSize = TextFieldAutoSize.LEFT;
stage.addChild(t);

// Register for capture phase
stage.addEventListener(MouseEvent.CLICK, clickListener, true);

// Register for target or bubbling phase
stage.addEventListener(MouseEvent.CLICK, clickListener, false);

// ...elsewhere in the class
private function clickListener (e:MouseEvent):void {
    var phase:String;
```
switch (e.eventPhase) {
  case EventPhase.CAPTURING_PHASE:
    phase = "Capture";
    break;

  case EventPhase.AT_TARGET:
    phase = "Target";
    break;

  case EventPhase.BUBBLING_PHASE:
    phase = "Bubbling";
    break;
  }
  trace("Current event phase: " + phase);
}

When the preceding code runs, if the user clicks the TextField object, the Flash runtime dispatches a MouseEvent.CLICK event targeted at the TextField object, and the output of the preceding code is:

Current event phase: Capture
Current event phase: Bubbling

(Remember, clickListener() registered with the Stage instance for both the capture phase and the bubbling phase, so it is triggered twice during event dispatches that target the Stage instance’s descendants.)

On the other hand, if the user clicks the display area, the Flash runtime dispatches a MouseEvent.CLICK event targeted at the Stage object, and the output of the preceding code is:

Current event phase: Target

As discussed in the next section, the eventPhase variable is typically used to differentiate between events targeted at an object and events targeted at that object’s descendants. Less commonly, the eventPhase variable is used to distinguish the capture phase from the bubbling phase within ancestor listeners that are registered for both of those phases.

**Distinguishing Events Targeted at an Object from Events Targeted at That Object’s Descendants**

When the eventPhase variable of the Event object passed to a listener function is set to EventPhase.AT_TARGET, we know that the event dispatch is targeted at the object with which the listener registered. On the other hand, when eventPhase is set to either EventPhase.CAPTURING_PHASE or EventPhase.BUBBLING_PHASE, we know that the event dispatch is targeted at a descendant of the object with which the listener registered.
Therefore, a listener can use the following code to ignore events targeted at descendants of the object with which it registered:

```javascript
private function someListener(e:SomeEvent):void {
    if (e.eventPhase == EventPhase.AT_TARGET) {
        // Code here is executed only when the object that registered this
        // listener is the event target.
    }
}
```

We can use the preceding technique to write code that responds to input received by a given object but not by any of its descendants. For example, imagine an application in which the Stage instance contains many buttons, text fields, and other input-receiving objects. To respond to mouse clicks when they occur over vacant areas of the Stage instance only, we use the following code:

```javascript
// Register with the Stage instance for MouseEvent.CLICK events.
// As a result, clickListener() will be invoked when *any* object
// on the display list is clicked.
stage.addEventListener(MouseEvent.CLICK, clickListener);

// ...elsewhere, define the MouseEvent.CLICK event listener
private function clickListener (e:MouseEvent):void {
    // If this listener was triggered during the target phase...
    if (e.eventPhase == EventPhase.AT_TARGET) {
        // ...then the Stage instance was clicked. Proceed with
        // "Stage instance was clicked" response code.
    }
}
```

To create an event listener that ignores events targeted at the object with which it registered, we use the following code:

```javascript
private function someListener(e:SomeEvent):void {
    if (e.eventPhase != EventPhase.AT_TARGET) {
        // Code here is executed only when the object that registered this
        // listener is a descendant of the event target.
    }
}
```

For example, the following listener responds to mouse clicks that occur over any object on the display list, but not over a vacant area of the Stage instance:

```javascript
// Register with the Stage instance for MouseEvent.CLICK events
stage.addEventListener(MouseEvent.CLICK, clickListener);

// ...elsewhere, define the MouseEvent.CLICK event listener
private function clickListener (e:SomeInputEvent):void {
    // If this listener was not triggered during the target phase...
    if (e.eventPhase != EventPhase.AT_TARGET) {
        // ...then the target must be a descendant of the Stage instance.
        // Hence, the mouse must have clicked some object on the display list
        // other than the Stage instance.
    }
}
```
Now let's move on to the next hierarchical-event-dispatch topic: stopping an event dispatch.

**Stopping an Event Dispatch**

At any point during the event flow, every event listener—including those registered with the target object and those registered with its ancestors—can put a stop to the entire event dispatch. Stopping an event's dispatch is referred to as **consuming the event**.

To stop an event dispatch, we invoke the `Event` class's instance method `stopImmediatePropagation()` or `stopPropagation()` on the `Event` object passed to a listener function. The `stopImmediatePropagation()` method stops the event dispatch immediately, without allowing any remaining listeners to be triggered; the `stopPropagation()` method stops the event dispatch after ActionScript triggers the remaining listeners registered with object currently being notified of the event.

For example, suppose we have a `Sprite`, container, that contains a `TextField`, child:

```actionscript
var container:Sprite = new Sprite();
var child:TextField = new TextField();
child.text = "click here";
child.autoSize = TextFieldAutoSize.LEFT;
container.addChild(child);
```

Further suppose we have three event listener functions: `containerClickListenerOne()`, `containerClickListenerTwo()`, and `childClickListener()`. Register `containerClickListenerOne()` and `containerClickListenerTwo()` with container for `MouseEvent.CLICK` event notification during the capture phase:

```actionscript
container.addEventListener(MouseEvent.CLICK,
    containerClickListenerOne,
    true);
container.addEventListener(MouseEvent.CLICK,
    containerClickListenerTwo,
    true);
```

Then we register `childClickListener()` with child for `MouseEvent.CLICK` event notification during the target phase:

```actionscript
child.addEventListener(MouseEvent.CLICK, childClickListener, false);
```

Under normal circumstances, when the user clicks child, all three event listeners are triggered—two during the capture phase, and one during the target phase. If, however, `containerClickListenerOne()` consumes the event using `stopImmediatePropagation()`, then neither `containerClickListenerTwo()` nor `childClickListener()` is triggered.

```actionscript
public function containerClickListenerOne (e:MouseEvent):void {
    // Prevent containerClickListenerTwo() and childClickListener() from
    // receiving the event
    e.stopImmediatePropagation();
}
```
On the other hand, if `containerClickListenerOne()` consumes the event using `stopPropagation()` rather than `stopImmediatePropagation()`, then `container`'s remaining `MouseEvent.CLICK` event listeners are triggered before the event dispatch stops. Hence, `containerClickListenerTwo()` receives the event, but `childClickListener()` does not.

```ActionScript
public function containerClickListenerOne (e:MouseEvent):void {
    // Prevent just childClickListener() from receiving the event
    e.stopPropagation();
}
```

Note that the preceding example relies on `containerClickListenerOne()` being registered before `containerClickListenerTwo()`. For information on the order in which event listeners are triggered, see Chapter 12.

Events are typically consumed in order to stop or override a program’s normal response to an event. For example, suppose a `Sprite` subclass, `ToolPanel`, contains a group of interface controls, each of which accepts user input. The `ToolPanel` class has two operational states: enabled and disabled. When a `ToolPanel` object is disabled, the user should not be able to interact with any of its nested interface controls.

To implement the “disabled” state, each `ToolPanel` object registers a method, `clickListener()`, for capture-phase `MouseEvent.CLICK` event notification. When a `ToolPanel` object is disabled, the `clickListener()` method stops all click events from reaching child `Tool` objects. Example 21-3 shows the `ToolPanel` class, greatly simplified to emphasize its event consumption code (shown in bold). In the example, the `ToolPanel` class’s child interface controls are instances of a generic `Tool` class that is not shown. In a real application, however, the controls might be buttons or menus or any other form of interactive tool.

**Example 21-3. Consuming an event**

```ActionScript
package {
    import flash.display.Sprite;
    import flash.events.*;

    public class ToolPanel extends Sprite {
        private var enabled:Boolean;

        public function ToolPanel () {
            enabled = false;

            var tool1:Tool = new Tool();
            var tool2:Tool = new Tool();
            var tool3:Tool = new Tool();

            tool2.x = tool1.width + 10;
            tool3.x = tool2.x + tool2.width + 10;

            addChild(tool1);
            addChild(tool2);
        }
    }
}
```
In typical application development, the `stopPropagation()` method is used much more frequently than the `stopImmediatePropagation()` method. Nevertheless, the `stopImmediatePropagation()` method is used in the following situations:

- When a target object wishes to prevent its own listeners from being triggered by an event
- When a program wishes to prevent all listeners from responding to a given event

Let’s consider an example for each of the preceding situations, starting with a target object that prevents its own listeners from being triggered by an event. Imagine a space-shooter game that includes the following classes:

- `GameManager`, a class that manages gameplay
- `PlayerShip`, a class that represents the player’s spacecraft

The `GameManager` defines a custom event, `GameManager.SHIP_HIT`, which is dispatched when an enemy’s missile hits the player’s ship. Each `GameManager.SHIP_HIT` event dispatch is targeted at the `PlayerShip` object. The `PlayerShip` object registers two listeners to respond to `GameManager.SHIP_HIT` events. One listener plays an explosion animation, and the other plays an explosion sound.

When the player dies, a new player ship is created, and that ship is invincible for five seconds. While the ship is invincible, the `PlayerShip` object’s `GameManager.SHIP_HIT` listeners should not play the “damaged-ship” animation or sound.

To prevent the `GameManager.SHIP_HIT` listeners from executing when the ship is invincible, the `PlayerShip` class registers a third listener, `hitListener()`, designed to consume `GameManager.SHIP_HIT` events based on the current ship status (invincible or not invincible). The `hitListener()` method is registered in `PlayerShip`’s constructor, with a priority of `int.MAX_VALUE`, as follows:

```javascript
Example 21-3. Consuming an event (continued)
addChild(tool3);

// Register with this object for MouseEvent.CLICK event notification
// during the capture phase
addEventListener(MouseEvent.CLICK, clickListener, true);
}

private function clickListener (e:MouseEvent):void {
    // If this ToolPanel object is disabled...
    if (!enabled) {
        // ...then stop this click event from reaching this ToolPanel
        // object’s descendants
        e.stopPropagation();
        trace("Panel disabled. Click event dispatch halted.");
    }
}
}

// Register with this object for MouseEvent.CLICK event notification
// during the capture phase
addEventListener(MouseEvent.CLICK, clickListener, true);
```

In typical application development, the `stopPropagation()` method is used much more frequently than the `stopImmediatePropagation()` method. Nevertheless, the `stopImmediatePropagation()` method is used in the following situations:

- When a target object wishes to prevent its own listeners from being triggered by an event
- When a program wishes to prevent all listeners from responding to a given event

Let’s consider an example for each of the preceding situations, starting with a target object that prevents its own listeners from being triggered by an event. Imagine a space-shooter game that includes the following classes:

- `GameManager`, a class that manages gameplay
- `PlayerShip`, a class that represents the player’s spacecraft

The `GameManager` defines a custom event, `GameManager.SHIP_HIT`, which is dispatched when an enemy’s missile hits the player’s ship. Each `GameManager.SHIP_HIT` event dispatch is targeted at the `PlayerShip` object. The `PlayerShip` object registers two listeners to respond to `GameManager.SHIP_HIT` events. One listener plays an explosion animation, and the other plays an explosion sound.

When the player dies, a new player ship is created, and that ship is invincible for five seconds. While the ship is invincible, the `PlayerShip` object’s `GameManager.SHIP_HIT` listeners should not play the “damaged-ship” animation or sound.

To prevent the `GameManager.SHIP_HIT` listeners from executing when the ship is invincible, the `PlayerShip` class registers a third listener, `hitListener()`, designed to consume `GameManager.SHIP_HIT` events based on the current ship status (invincible or not invincible). The `hitListener()` method is registered in `PlayerShip`’s constructor, with a priority of `int.MAX_VALUE`, as follows:
public class PlayerShip {
    public function PlayerShip () {
        addEventListener(GameManager.HIT, hitListener, false, int.MAX_VALUE);
    } 
}

In Chapter 12 we learned that, by default, an object’s event listeners are triggered according to the order in which they were registered. We also learned that this default “trigger order” can be superceded by addEventListener()’s priority parameter.

Because hitListener() is registered in the PlayerShip constructor with the highest possible priority, it is always the first of PlayerShip’s GameManager.SHIP_HIT listeners executed. When the PlayerShip object is not invincible, and a GameManager.SHIP_HIT event occurs, hitListener() does nothing. But when the PlayerShip object is invincible, and a GameManager.SHIP_HIT event occurs, hitListener() first consumes the event, and then dispatches a new event, PlayerShip.HIT_DEFLECTED. Listeners registered for PlayerShip.HIT_DEFLECTED then play a special animation and sound indicating that the ship was not damaged.

The code for hitListener() follows; notice the use of the stopImmediatePropagation() method:

    private function hitListener (e:Event):void {
        if (invincible) {
            // Prevent other PlayerShip listeners from receiving event notification
            e.stopImmediatePropagation();
            // Broadcast a new event
            dispatchEvent(new Event(PlayerShip.HIT_DEFLECTED, true));
        }
    }

In the preceding PlayerShip scenario, note that although the PlayerShip object can prevent its own GameManager.SHIP_HIT listeners from being triggered, it cannot prevent GameManager.SHIP_HIT listeners registered with its display ancestors from being triggered. Specifically, any listeners registered for the capture phase with the PlayerShip object’s display ancestors will always be notified of the GameManager.SHIP_HIT event even if the PlayerShip object consumes it. However, after the PlayerShip object consumes the GameManager.SHIP_HIT event, its ancestors do not receive notification during the bubbling phase.

Now let’s turn to the second stopImmediatePropagation() scenario, in which a program wishes to prevent all listeners from responding to a given event. Suppose we’re writing a set of user-interface components that automatically change to an inactive state when the Flash runtime loses operating-system focus, and change to an active state when the Flash runtime gains operating-system focus. In order to detect the gaining or losing of operating system focus, our components register listeners for the built-in events Event.ACTIVATE and Event.DEACTIVATE (for details on Event.ACTIVATE and Event.DEACTIVATE, see Chapter 22).
Now further suppose we’re writing a testing application to stress test our components. Our stress-test application programmatically triggers the components’ interactive behavior. In our test, we must ensure that the built-in Event.DEACTIVATE event does not make the test components inactive; otherwise, our test application will not be able to trigger them programmatically. Hence, in our test application’s main class constructor, we register an Event.DEACTIVATE listener with the Stage instance. That listener uses stopImmediatePropagation() to consume all built-in Event.DEACTIVATE events, as follows.

```javascript
private function deactivateListener (e: Event):void {
    e.stopImmediatePropagation();
}
```

Because our test application consumes all Event.DEACTIVATE events, the components never receive Event.DEACTIVATE notifications, and, hence, never become inactive in response to the Flash runtime losing system focus. The administrator of the test application can then focus and defocus the Flash runtime while the test runs without interfering with the test application’s ability to control the components programmatically.

## Event Priority and the Event Flow

When an event listener registers with an object in a display hierarchy, the priority parameter affects the trigger order of listeners registered with that object only. The priority parameter does not, and cannot, alter the order in which objects in the event flow are notified.

There is no way to force a listener of one object in the event flow to be triggered before or after a listener of another object in the same event flow.

For example, suppose a Sprite object contains a TextField object. The Sprite object registers a MouseEvent.CLICK listener, spriteClickListener(), with useCapture set to false and priority set to 2:

```javascript
theSprite.addEventListener(MouseEvent.CLICK, spriteClickListener, false, 2)
```

Likewise, the TextField object registers a MouseEvent.CLICK listener, textClickListener(), with useCapture set to false and priority set to 1:

```javascript
theTextField.addEventListener(MouseEvent.CLICK, textClickListener, false, 1)
```

When the user clicks the TextField object, the Flash runtime dispatches a MouseEvent.CLICK event targeted at the TextField object. In response, textClickListener() is triggered during the target phase, before spriteClickListener(), which is triggered during the bubbling phase. The two event listeners are triggered
according to the order of the event flow, even though spriteClickListener() registered with a higher priority than textClickListener().

For more information on event priority, see Chapter 12.

## Display-Hierarchy Mutation and the Event Flow

Immediately before ActionScript dispatches a given event, it predetermines the entire event flow for that dispatch based on the current state of the event target’s display hierarchy. That is, the list of objects that will be notified of the event, and the order in which those objects will be notified, is predetermined before the event dispatch begins. Once the event dispatch begins, objects are notified of the event according to that predetermined order—even if the structure of the target object’s display hierarchy is modified by event listeners during the event dispatch.

For example, suppose we have a TextField instance contained by a Sprite instance that is, itself, contained by the Stage instance. Further suppose we register a listener, stageClickListener(), with the Stage instance for capture-phase MouseEvent.CLICK event notification, as shown in the following code:

```actionscript
stage.addEventListener(MouseEvent.CLICK, stageClickListener, true);
```

Finally, suppose that the registered stageClickListener() function contains code that removes the TextField object from its Sprite object parent, as follows:

```actionscript
private function stageClickListener (e:MouseEvent):void {
    // If the TextField object was clicked...
    if (e.target == textField) {
        // ...remove it
        removeChild(textField);
        textField = null;
    }
}
```

When the user clicks the text field, ActionScript dispatches a MouseEvent.CLICK event targeted at the TextField object. Before the dispatch begins, ActionScript predetermines the entire event flow, as follows:

**CAPTURE PHASE:**
1. Stage object
2. Sprite object

**TARGET PHASE:**
3. TextField object

**BUBBLING PHASE:**
4. Sprite object
5. Stage object

When the event dispatch begins, ActionScript first notifies the Stage object of the event (1). That notification triggers the Stage object’s listener, stageClickListener(), which removes the TextField object from the display list. Next, even though the Sprite object is now no longer an ancestor of the TextField object, ActionScript notifies the Sprite object of the event (2). Then, even though the TextField object is no longer on the display list, ActionScript notifies the TextField object of the event (3).
Finally, during the bubbling phase, ActionScript again notifies the *Sprite* object (4) and the *Stage* object (5) of the event. Even though the display hierarchy containing the event target was modified during the event dispatch, the event still propagates through the entire predetermined event flow.

Once the event flow is established for a given event dispatch, it is fixed for the duration of that dispatch.

Example 21-4 shows the code for the preceding scenario in the context of an example class, *DisappearingTextField*. In the example, an instance of the custom class *DisappearingTextField* (a *Sprite* subclass) plays the role of the preceding scenario’s *Sprite* object.

**Example 21-4. The immutable event flow**

```javascript
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;

    public class DisappearingTextField extends Sprite {
        private var textField:TextField;
        public function DisappearingTextField () {
            // Create the TextField object
            textField = new TextField();
            textField.text = "Click here";
            textField.autoSize = TextFieldAutoSize.LEFT;

            // Add the TextField object to the DisappearingTextField object
            addChild(textField);

            // Register with the Stage instance for MouseEvent.CLICK events
            stage.addEventListener(MouseEvent.CLICK, stageClickListener, true);

            // To prove that the TextField object receives MouseEvent.CLICK event
            // notification even after it is removed from this
            // DisappearingTextField instance, we register a listener with the
            // TextField object for MouseEvent.CLICK events
            textField.addEventListener(MouseEvent.CLICK, textFieldClickListener);
        }

        // This function runs when the user clicks any object on the
        // display list
        private function stageClickListener (e:MouseEvent):void {
            // If the TextField object was clicked...
            if (e.target == textField) {
                // ...remove it
                removeChild(textField);
                textField = null;
            }
        }

        private function textFieldClickListener (e:MouseEvent):void {
            // Remove the TextField object from this DisappearingTextField instance
            removeChild(textField);
            textField = null;
        }
    }
}
```

524 | Chapter 21: Events and Display Hierarchies
As we’ve just learned, when a given event occurs, ActionScript notifies the appropriate objects according to the predetermined order of the event flow. In turn, when each object receives notification of the event, its event listeners are triggered. The specific list of listeners that are triggered by a given event notification is determined immediately before that notification occurs. Once the notification has begun, that listener list cannot be altered.

For example, consider a `MouseEvent.CLICK` event dispatch targeted at a `Sprite` object that is contained by the `Stage` instance. The event flow comprises three event notifications, as follows:

- **CAPTURE PHASE**: (1) Stage object notified
- **TARGET PHASE**: (2) Sprite object notified
- **BUBBLING PHASE**: (3) Stage object notified

Suppose that, during the first notification (1), code in a `Stage` listener registers a new `MouseEvent.CLICK` listener with the `Sprite` object. Because the event has not yet propagated to the `Sprite` object, the new listener will be triggered during the second notification (2).

Now further suppose that, during the first notification (1), code in a `Stage` listener registers a new `MouseEvent.CLICK` listener with the `Stage` instance, for bubbling-phase notification. Because the first notification (1) has already begun, the `Stage` instance’s list of listeners has already been frozen, so the new listener is not triggered during the first notification (1). However, the new listener is triggered later in the event flow, during the third notification (3).

Finally, suppose that, during the second notification (2), code in a `Sprite` listener unregisters an existing `MouseEvent.CLICK` listener from the `Sprite` object. Because the second notification (2) has already begun, its list of listeners has already been frozen, so the removed listener is still triggered during the second notification. Of course, if another `MouseEvent.CLICK` event were dispatched, the removed listener would not be triggered.

At any point during a given event dispatch, the list of listeners being triggered during the current notification cannot be modified, but the list of listeners to be triggered by notifications that occur later in the event flow can be modified.
Custom Events and the Event Flow

ActionScript’s hierarchical event system applies to all event-dispatches targeted display objects—even those event dispatches instigated manually by the programmer. When a custom event dispatch targets an object in a display hierarchy, the ancestors of that target object are notified of the event.

The generalized code shown in Example 21-5 demonstrates how, just like built-in events, custom events propagate through the event flow. In the example, a test class, CustomEventDemo, instructs ActionScript to dispatch a custom event targeted at a Sprite object on the display list.

Example 21-5. A custom event dispatched through the event flow

```actionscript
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;

    public class CustomEventDemo extends Sprite {
        public static const SOME_EVENT:String = "SOME_EVENT";

        public function CustomEventDemo () {
            var sprite:Sprite = new Sprite();
            addChild(sprite);

            // Register someEventListener() with the Stage instance for
            // CustomEventDemo.SOME_EVENT notification.
            stage.addEventListener(CustomEventDemo.SOME_EVENT, someEventListener);

            // Dispatch a CustomEventDemo.SOME_EVENT event to an object on
            // the display list. Set the Event constructor's second parameter
            // to true so the event bubbles.
            sprite.dispatchEvent(new Event(CustomEventDemo.SOME_EVENT, true));
        }

        private function someEventListener (e:Event):void {
            trace("SOME_EVENT occurred.");
        }
    }
}
```

In response to the dispatchEvent() invocation in Example 21-5, ActionScript dispatches a CustomEventDemo.SOME_EVENT event into the event flow, targeted at sprite. The event flow is as follows:

```
Stage instance
    |-> CustomEventDemo object
        |-> Sprite object
```
During the capture phase, the CustomEventDemo.SOME_EVENT event dispatch propagates from the Stage instance to the CustomEventDemo object. During the target phase, the event propagates to the Sprite object. Finally, during the bubbling phase, the event propagates back to the CustomEventDemo object and then back to the Stage instance. When the Stage instance receives event notification during the bubbling phase, someEventListener() is triggered. Even though CustomEventDemo.SOME_EVENT is a custom event, it still propagates through the event flow.

As with the built-in events, ActionScript’s hierarchical event architecture can help centralize code that responds to custom events. For example, suppose we’re building an online ordering system with a shopping-basket widget that contains selectable product icons. The shopping-basket widget is represented by an instance of the custom class, ShoppingBasket. Likewise, each product icon is represented by an instance of the custom class, Product. The Product instances are display children of the ShoppingBasket instance. The ShoppingBasket instance has a title bar that displays the name of the currently selected product.

When the user selects a product icon in the shopping-basket widget, our application dispatches a custom Product.PRODUCT_SELECTED event targeted at the corresponding Product instance. Because the ShoppingBasket instance is an ancestor of all Product instances, it is notified every time the Product.PRODUCT_SELECTED event is dispatched. Hence, to keep the ShoppingBasket instance’s title bar synchronized with the currently selected product, we simply register a single Product.PRODUCT_SELECTED listener—productSelectedListener()—with the ShoppingBasket instance. When productSelectedListener() is triggered, we know a product has been selected, so we update the shopping-basket’s title bar with the name of the newly selected product.

Examples 21-6 and 21-7 show the ShoppingBasket and Product classes. The comments will guide you through the code. Event-related sections are shown in bold.

*Example 21-6. The ShoppingBasket class*

```actionscript
package {    import flash.display.*;    import flash.text.*;    import flash.events.*;

// An online shopping basket that can visually contain Product objects.    public class ShoppingBasket extends Sprite {        // The on-screen title bar for the shopping basket panel        private var title:TextField;        // An array of the products in this basket        private var products:Array;        // The currently selected product        private var selectedProduct:Product;

        // Constructor        public function ShoppingBasket () {            // Create a new empty array to hold Product objects            products = new Array();
```
Example 21-6. The ShoppingBasket class (continued)

// Create the on-screen title bar
    title = new TextField();
    title.text = "No product selected";
    title.autoSize = TextFieldAutoSize.LEFT;
    title.border = true;
    title.background = true;
    title.selectable = false;
    addChild(title);

    // Start listening for Product.PRODUCT_SELECTED event dispatches
    // targeted at child Product objects.
    addEventListener(Product.PRODUCT_SELECTED, productSelectedListener);
}

// Adds a new Product object to the shopping basket
public function addProduct (product:Product):void {
    // Create a new product and add it to the products array
    products.push(product);
    // Add the new product to this object's display hierarchy
    addChild(products[products.length-1]);

    // Now that there's a new product, reposition all products
    updateLayout();
}

// A very simple product-layout algorithm that sorts all products
// into a single horizontal line in the order they were added, from left
// to right
public function updateLayout ():void {
    var totalX:Number = 0;
    for (var i:int = 0; i < products.length; i++) {
        products[i].x = totalX;
        totalX += products[i].width + 20;     // 20 is the gutter width
        products[i].y = title.height + 20;    // 20 is the gutter height
    }
}

// Responds to Product.PRODUCT_SELECTED event dispatches targeted at
// child Product objects. When a product is selected, this method
// updates the shopping basket's title bar to match the selected
// product's name.
private function productSelectedListener (e:Event):void {
    // Remember the selected product in case we need to reference it
    // in future
    selectedProduct = Product(e.target);

    // Update the shopping basket's title
    title.text = selectedProduct.getName();
}
Example 21-7 shows the code for the *Product* class.

**Example 21-7. The Product class**

```ActionScript
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;

    // A product icon that can be placed in a ShoppingBasket object using
    // ShoppingBasket.addProduct(). In this simplified version, the icon is
    // simply a text field with no corresponding graphical icon.
    public class Product extends Sprite {
        public static const PRODUCT_SELECTED:String = "PRODUCT_SELECTED";
        // The on-screen label showing the product's name
        private var label:TextField;
        // Constructor
        public function Product (productName:String) {
            this.productName = productName;
            label = new TextField();
            label.text = productName;
            label.autoSize = TextFieldAutoSize.LEFT;
            label.border = true;
            label.background = true;
            label.selectable = false;
            addChild(label);
            // Start listening for mouse clicks. By registering for
            // MouseEvent.CLICK events with this object, we'll receive
            // notification any time its children (e.g., the label) are clicked.
            addEventListener(MouseEvent.CLICK, clickListener);
        }

        // Returns the product's name
        public function getName ():String {
            return productName;
        }

        // Handles MouseEvent.CLICK events. In this example, simply clicking a
        // product selects it, and causes the Product.PRODUCT_SELECTED event to
        // be dispatched. In a more complete implementation, other factors
        // might be involved. For example, products might be selectable
        // via the keyboard, and selection might be disabled during a
        // transaction with the server.
        private function clickListener (e:MouseEvent):void {
            // Notify all registered listeners that this product was selected.
            // Thanks to ActionScript's hierarchical event dispatch system,
            // by dispatching a custom event targeted at this object, we trigger
        }
    }
}
```

Example 21-8 presents a very simple application demonstrating the basic usage of the `ShoppingBasket` and `Product` classes from Example 21-6 and Example 21-7.

```asciidoc
Example21-8.A ShoppingBasket demo
package {
    import flash.display.Sprite;

    public class ShoppingBasketDemo extends Sprite {
        public function ShoppingBasketDemo () {
            var basket:ShoppingBasket = new ShoppingBasket();
            basket.addProduct(new Product("Nintendo Wii"));
            basket.addProduct(new Product("Xbox 360"));
            basket.addProduct(new Product("PlayStation 3"));
            addChild(basket);
        }
    }
}
```

**On to Input Events**

We’ve now studied pretty much everything there is to know about the core workings of ActionScript’s hierarchical event system. In the next chapter, we’ll put our new theoretical knowledge into practice as we explore Flash Player’s built-in user-input events.
### Other resources from O’Reilly

#### Related titles
- ActionScript 3.0 Design Patterns
- Dynamic HTML: The Definitive Reference
- Ajax on Java
- Ajax on Rails
- Learning JavaScript
- Programming Atlas
- Head Rush Ajax
- Rails Cookbook

#### oreilly.com
Oreilly.com is more than a complete catalog of O’Reilly books. You’ll also find links to news, events, articles, weblogs, sample chapters, and code examples.

#### oreillynet.com
Oreilly Net is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

#### Conferences
O’Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator’s knowledge into useful skills for those in the trenches. Visit conferences.oreilly.com for our upcoming events.

#### Safari Bookshelf
Safari Bookshelf (safari.oreilly.com) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.