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.
# 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>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.</td>
<td>Compiling and Running a Program</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>Compiling with the Flash Authoring Tool</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>Compiling with Flex Builder 2</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>Compiling with mxmlc</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Compiler Restrictions</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>The Compilation Process and the Classpath</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>Strict-Mode Versus Standard-Mode Compilation</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>The Fun’s Not Over</td>
<td>136</td>
</tr>
<tr>
<td>8.</td>
<td>Datatypes and Type Checking</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>Datatypes and Type Annotations</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>Untyped Variables, Parameters, Return Values, and Expressions</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>Strict Mode’s Three Special Cases</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>Warnings for Missing Type Annotations</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>Detecting Reference Errors at Compile Time</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>Casting</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>Conversion to Primitive Types</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Default Variable Values</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>null and undefined</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>Datatypes in the Virtual Zoo</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>More Datatype Study Coming Up</td>
<td>158</td>
</tr>
<tr>
<td>9.</td>
<td>Interfaces</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>The Case for Interfaces</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>Interfaces and Multidatatype Classes</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>Interface Syntax and Use</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>Another Multiple-Type Example</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>More Essentials Coming</td>
<td>171</td>
</tr>
<tr>
<td>10.</td>
<td>Statements and Operators</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>Statements</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>Operators</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>Up Next: Managing Lists of Information</td>
<td>185</td>
</tr>
</tbody>
</table>
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
Garbage Collection Demonstration 277
On to ActionScript Backcountry 278

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
# 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
24. Programmatic Animation ........................................ 610
   No Loops .......................................................... 610
   Animating with the ENTER_FRAME Event .................... 611
   Animating with the TimerEvent.TIMER Event ............... 616
   Choosing Between Timer and Event.ENTER_FRAME .......... 623
   A Generalized Animator ........................................ 624
   Velocity-Based Animation ..................................... 627
   Moving On to Strokes ‘n’ Fills ................................ 628

25. Drawing with Vectors .......................................... 629
   Graphics Class Overview ........................................ 629
   Drawing Lines ..................................................... 630
   Drawing Curves ................................................... 633
   Drawing Shapes ................................................... 634
   Removing Vector Content ...................................... 636
   Example: An Object-Oriented Shape Library ................. 637
   From Lines to Pixels ............................................. 647

26. Bitmap Programming ........................................... 648
   The BitmapData and Bitmap Classes ......................... 649
   Pixel Color Values .............................................. 649
   Creating a New Bitmap Image .................................. 654
   Loading an External Bitmap Image ......................... 656
   Examining a Bitmap ............................................. 658
   Modifying a Bitmap ............................................. 664
   Copying Graphics to a BitmapData Object .................. 672
   Applying Filters and Effects .................................. 686
   Freeing Memory Used by Bitmaps ................................ 694
   Words, Words, Words ........................................... 695

27. Text Display and Input ........................................ 696
   Creating and Displaying Text ................................ 699
   Modifying a Text Field’s Content ............................ 705
   Formatting Text Fields ......................................... 708
   Fonts and Text Rendering ...................................... 735
   Missing Fonts and Glyphs ...................................... 748
   Determining Font Availability ................................ 749
   Determining Glyph Availability ............................... 751
   Embedded-Text Rendering ...................................... 752
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Field Input</td>
<td>755</td>
</tr>
<tr>
<td>Text Fields and the Flash Authoring Tool</td>
<td>759</td>
</tr>
<tr>
<td>Loading…Please Wait…</td>
<td>761</td>
</tr>
<tr>
<td><strong>28. Loading External Display Assets</strong></td>
<td>762</td>
</tr>
<tr>
<td>Using Loader to Load Display Assets at Runtime</td>
<td>763</td>
</tr>
<tr>
<td>Compile-Time Type-Checking for Runtime-Loaded Assets</td>
<td>781</td>
</tr>
<tr>
<td>Accessing Assets in Multiframe .swf Files</td>
<td>790</td>
</tr>
<tr>
<td>Instantiating a Runtime-Loaded Asset</td>
<td>793</td>
</tr>
<tr>
<td>Using Socket to Load Display Assets at Runtime</td>
<td>796</td>
</tr>
<tr>
<td>Removing Runtime Loaded .swf Assets</td>
<td>806</td>
</tr>
<tr>
<td>Embedding Display Assets at CompileTime</td>
<td>807</td>
</tr>
<tr>
<td>On to Part III</td>
<td>818</td>
</tr>
<tr>
<td><strong>Part III. Applied ActionScript Topics</strong></td>
<td></td>
</tr>
<tr>
<td><strong>29. ActionScript and the Flash Authoring Tool</strong></td>
<td>821</td>
</tr>
<tr>
<td>The Flash Document</td>
<td>821</td>
</tr>
<tr>
<td>Timelines and Frames</td>
<td>822</td>
</tr>
<tr>
<td>Timeline Scripting</td>
<td>826</td>
</tr>
<tr>
<td>The Document Class</td>
<td>828</td>
</tr>
<tr>
<td>Symbols and Instances</td>
<td>832</td>
</tr>
<tr>
<td>Linked Classes for Movie Clip Symbols</td>
<td>834</td>
</tr>
<tr>
<td>Accessing Manually Created Symbol Instances</td>
<td>838</td>
</tr>
<tr>
<td>Accessing Manually Created Text</td>
<td>844</td>
</tr>
<tr>
<td>Programmatic Timeline Control</td>
<td>845</td>
</tr>
<tr>
<td>Instantiating Flash Authoring Symbols via ActionScript</td>
<td>847</td>
</tr>
<tr>
<td>Instance Names for Programmatically Created Display Objects</td>
<td>848</td>
</tr>
<tr>
<td>Linking Multiple Symbols to a Single Superclass</td>
<td>849</td>
</tr>
<tr>
<td>The Composition-Based Alternative to Linked Classes</td>
<td>851</td>
</tr>
<tr>
<td>Preloading Classes</td>
<td>852</td>
</tr>
<tr>
<td>Up Next: Using the Flex Framework</td>
<td>855</td>
</tr>
<tr>
<td>30. A Minimal MXML Application</td>
<td>856</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>The General Approach</td>
<td>856</td>
</tr>
<tr>
<td>A Real UI Component Example</td>
<td>859</td>
</tr>
<tr>
<td>Sharing with Your Friends</td>
<td>860</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31. Distributing a Class Library</th>
<th>861</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing Class Source Files</td>
<td>862</td>
</tr>
<tr>
<td>Distributing a Class Library as a .swc File</td>
<td>863</td>
</tr>
<tr>
<td>Distributing a Class Library as a .swf File</td>
<td>867</td>
</tr>
<tr>
<td>But Is It Really Over?</td>
<td>873</td>
</tr>
</tbody>
</table>

**Appendix** | 875 |

**Index** | 891 |
One of the primary activities of ActionScript programming is displaying things on the screen. Accordingly, the Flash platform provides a wide range of tools for creating and manipulating graphical content. These tools can be broken into two general categories:

- The Flash runtime display API, a set of classes for working with interactive visual objects, bitmaps, and vector content
- Ready-made user interface components:
  - The Flex framework’s UI component set, a sophisticated collection of customizable user-interface widgets built on top of the display API
  - The Flash authoring tool’s UI component set, a collection of user-interface widgets with a smaller file size, lower memory usage, and fewer features than Flex framework’s UI component set

The display API is built directly into all Flash runtimes and is, therefore, available to all .swf files. The display API is designed for producing highly customized user interfaces or visual effects, such as those often found in motion graphics and games. This chapter focuses entirely on the display API.

The Flex framework’s UI component set is part of the Flex framework, an external class library included with Adobe Flex Builder and also available in standalone form for free at: http://www.adobe.com/go/flex2_sdk. The Flex framework’s UI component set is designed for building applications with relatively standard user interface controls (scrollbars, pull-down menus, data grids, etc.). The Flex framework’s interface widgets are typically used in MXML applications, but can also be included in primarily ActionScript-based applications. For details on using the Flex framework in ActionScript, see Chapter 30.

The Flash authoring tool’s UI component set is designed for use with .swf files created in the Flash authoring tool, and for situations where file size and low memory usage are more important than advanced component features such as data binding and advanced styling options. The Flash authoring tool’s UI component set and the
Flex framework’s UI component set share a very similar API, allowing developers to reuse knowledge when moving between the two component sets.

In Flash Player 8 and older, ActionScript provided the following four basic building blocks for creating and managing visual content:

**Movie clip**
- A container for graphical content, providing interactivity, primitive drawing, hierarchical layout, and animation feature

**Text field**
- A rectangular region containing a formatted text

**Button**
- An input control representing a very simple interactive “push button”

**Bitmap (introduced in Flash Player 8)**
- A graphic in bitmap-format

The preceding items continue to be available in the display API, but the classes representing them in ActionScript 3.0 (MovieClip, TextField, SimpleButton, and Bitmap) have been enhanced and revised, and situated logically within a larger context.

## Display API Overview

In ActionScript, all graphical content is created and manipulated using the classes in the display API. Even the interface widgets in the Flex framework and Flash authoring tool component sets use the display API as a graphical foundation. Many display API classes directly represent a specific type of on-screen graphical content. For example, the Bitmap class represents bitmap graphics, the Sprite class represents interactive graphics, and the TextField class represents formatted text. For the purposes of discussion, we’ll refer to classes that directly represent on-screen content (and superclasses of such classes) as **core display classes**. The remaining classes in the display API define supplementary graphical information and functionality but do not, themselves, represent on-screen content. For example, the CapStyle and JointStyle classes define constants representing line-drawing preferences, while the Graphics and BitmapData classes define a variety of primitive drawing operations. We’ll refer to these nondisplay classes as **supporting display classes**. Whether core or supporting, most of the display API classes reside in the package flash.display.

The core display classes, shown in Figure 20-1, are arranged in a class hierarchy that reflects three general tiers of functionality: display, user interactivity, and containment. Accordingly, the three central classes in the display API are: DisplayObject, InteractiveObject, and DisplayObjectContainer. Those three classes cannot be instantiated directly but rather provide abstract functionality that is applied by various concrete subclasses.
As discussed in Chapter 6, ActionScript 3.0 does not support true abstract classes. Hence, in Figure 20-1, DisplayObject, InteractiveObject, and DisplayObjectContainer are listed not as abstract classes, but as abstract-style classes. However, despite this technicality, for the sake of brevity in the remainder of this chapter, we’ll use the shorter term “abstract” when referring to the architectural role played by DisplayObject, InteractiveObject, and DisplayObjectContainer.

DisplayObject, the root of the core-display class hierarchy, defines the display API’s first tier of graphical functionality: on-screen display. All classes that inherit from DisplayObject gain a common set of fundamental graphical characteristics and capabilities. For example, every descendant of DisplayObject can be positioned, sized, and rotated with the variables x, y, width, height, and rotation. More than just a simple base class, DisplayObject is the source of many sophisticated capabilities in the display API, including (but not limited to):

- Converting coordinates (see the DisplayObject class’s instance methods localToGlobal() and globalToLocal() in Adobe’s ActionScript Language Reference)
- Checking intersections between objects and points (see the DisplayObject class’s instance methods hitTestObject() and hitTestPoint() in Adobe’s ActionScript Language Reference)
• Applying filters, transforms, and masks (see the DisplayObject class’s instance variables filters, transform, and mask in Adobe’s ActionScript Language Reference)
• Scaling disproportionately for “stretchy” graphical layouts (see the DisplayObject class’s instance variable scale9grid in Adobe’s ActionScript Language Reference)

Note that this book occasionally uses the informal term “display object” to mean any instance of a class descending from the DisplayObject class.

DisplayObject’s direct concrete subclasses—Video, Bitmap, Shape, MorphShape, and StaticText—represent the simplest type of displayable content: basic on-screen graphics that cannot receive input or contain nested visual content. The Video class represents streaming video. The Bitmap class renders bitmap graphics created and manipulated with the supporting BitmapData class. The Shape class provides a simple, lightweight canvas for vector drawing. And the special MorphShape and StaticText classes represent, respectively, shape tweens and static text created in the Flash authoring tool. Neither MorphShape nor StaticText can be instantiated with ActionScript.

DisplayObject’s only abstract subclass, InteractiveObject, establishes the second tier of functionality in the display API: interactivity. All classes that inherit from InteractiveObject gain the ability to respond to input events from the user’s mouse and keyboard. InteractiveObject’s direct concrete subclasses—TextField and SimpleButton—represent two distinct kinds of interactive graphical content. The TextField class represents a rectangular area for displaying formatted text and receiving text-based user input. The SimpleButton class represents Button symbols created in the Flash authoring tool and can also quickly create interactive buttons via ActionScript code. By responding to the input events broadcast by the TextField or SimpleButton, the programmer can add interactivity to an application. For example, a TextField instance can be programmed to change background color in response to a FocusEvent.FOCUS_IN event, and a SimpleButton instance can be programmed to submit a form in response to a MouseEvent.CLICK event.

InteractiveObject’s only abstract subclass, DisplayObjectContainer, is the base of the third and final functional tier in the display API: containment. All classes that inherit from DisplayObjectContainer gain the ability to physically contain any other DisplayObject instance. Containers are used to group multiple visual objects so they can be manipulated as one. Any time a container is moved, rotated, or transformed, the objects it contains inherit that movement, rotation, or transformation. Likewise, any time a container is removed from the screen, the objects it contains are removed with it. Furthermore, containers can be nested within other containers to create hierarchical groups of display objects. When referring to the objects in a display hierarchy, this book use standard tree-structure terminology; for example, an object that contains another object in a display hierarchy is referred to as that object’s parent, while the contained object is referred to as the parent’s child. In a multilevel display
hierarchy, the objects above a given object in the hierarchy are referred to as the object’s ancestors. Conversely, the objects below a given object in the hierarchy are referred to as the object’s descendants. Finally, the top-level object in the hierarchy (the object from which all other objects descend) is referred to as the root object.

Don’t confuse the ancestor objects and descendant objects in a display hierarchy with the ancestor classes and descendant classes in an inheritance hierarchy. For clarity, this book occasionally uses the terms “display ancestors” and “display descendants” when referring to ancestor objects and descendant objects in a display hierarchy.

DisplayObjectContainer’s subclasses—Sprite, MovieClip, Stage, and Loader—each provide a unique type of empty containment structure, waiting to be filled with content. Sprite is the centerpiece of the container classes. As a descendant of both the InteractiveObject the DisplayObjectContainer classes, Sprite provides the perfect foundation for building custom user interface elements from scratch. The MovieClip class is an enhanced type of Sprite that represents animated content created in the Flash authoring tool. The Stage class represents the Flash runtime’s main display area (the viewable region within the borders of the application window). Finally, the Loader class is used to load external graphical content locally or over the Internet.

Prior to ActionScript 3.0, the MovieClip class was used as an all-purpose graphics container (much like ActionScript 3.0’s Sprite class is used). As of ActionScript 3.0, MovieClip is used only to control instances of movie clip symbols created in the Flash authoring tool. Because ActionScript 3.0 does not provide a way to create timeline elements such as frames and tweens, there is no need to create new empty movie clips at runtime in ActionScript 3.0. Instead, all programmatically created graphics should be instances of the appropriate core display class (Bitmap, Shape, Sprite, TextField, etc.).

The display API provides a vast amount of functionality, dispersed over hundreds of methods and variables. While this book covers many of them, our focus in the coming chapters is on fundamental concepts rather than methodical coverage of each method and variable. For a dictionary-style reference to the display API, see Adobe’s ActionScript Language Reference.

Extending the Core-Display Class Hierarchy

While in many cases, the core display classes can productively be used without any modification, most nontrivial programs extend the functionality of the core display classes by creating subclasses suited to a custom purpose. For example, a geometric drawing program might define Ellipse, Rectangle, and Triangle classes that extend the Shape class. Similarly, a news viewer might define a Heading class that extends
TextField, and a racing game might define a Car class that extends Sprite. In fact, the user interface widgets in the Flex framework are all descendants of the Sprite class. In the chapters ahead, we’ll encounter many examples of custom display classes. As you learn more about the core display classes, start thinking about how you could add to their functionality; ActionScript programmers are expected and encouraged to expand and enhance the core display classes with custom code. For more information, see the section “Custom Graphical Classes,” later in this chapter.

The Display List

As we’ve just discussed, the core display classes represent the types of graphical content available in ActionScript. To create actual graphics from those theoretical types, we create instances of the core display classes and then add those instances to the display list. The display list is the hierarchy of all graphical objects currently displayed by the Flash runtime. When a display object is added to the display list and is positioned in a visible area, the Flash runtime renders that display object’s content to the screen.

The root of the display list is an instance of the Stage class, which is automatically created when the Flash runtime starts. This special, automatically created Stage instance serves two purposes. First, it acts as the outermost container for all graphical content displayed in the Flash runtime (i.e., it is the root of the display list). Second, it provides information about, and control over, the global characteristics of the display area. For example, the Stage class’s instance variable quality indicates the rendering quality of all displayed graphics; scaleMode indicates how graphics scale when the display area is resized; and frameRate indicates the current preferred frames per second for all animations. As we’ll see throughout this chapter, the Stage instance is always accessed relative to some object on the display list via the DisplayObject class’s instance variable stage. For example, if output_txt is a TextField instance currently on the display list, then the Stage instance can be accessed using output_txt.stage.
Prior to ActionScript 3.0, the `Stage` class did not contain objects on the display list. Furthermore, all `Stage` methods and variables were accessed via the `Stage` class directly, as in:

```
trace(Stage.align);
```

In ActionScript 3.0, `Stage` methods and variables are not accessed through the `Stage` class, and there is no global point of reference to the `Stage` instance. In ActionScript 3.0, the preceding line of code causes the following error:

```
Access of possibly undefined property 'align' through a reference with static type 'Class'
```

To avoid that error, access the `Stage` instance using the following approach:

```
trace(someDisplayObj.stage.align);
```

where `someDisplayObj` is an object currently on the display list. ActionScript 3.0's `Stage` architecture allows for the future possibility of multiple `Stage` instances and also contributes to Flash Player’s security (because unauthorized externally-loaded objects have no global point of access to the `Stage` instance).

Figure 20-2 depicts the state of the display list for an empty Flash runtime before any `.swf` file has been opened. The left side of the figure shows a symbolic representation of the Flash runtime, while the right side shows the corresponding display list hierarchy. When the Flash runtime is empty, the display list hierarchy contains one item only (the lone `Stage` instance). But we’ll soon add more!

![Figure 20-2. The display list for an empty Flash runtime](image)

When an empty Flash runtime opens a new `.swf` file, it locates that `.swf` file’s main class, creates an instance of it, and adds that instance to the display list as the `Stage` instance’s first child.

Recall that a `.swf` file’s main class must inherit from either `Sprite` or `MovieClip`, both of which are descendants of `DisplayObject`. Techniques for specifying a `.swf` file’s main class are covered in Chapter 7.
The .swf file’s main class instance is both the program entry point and the first visual object displayed on screen. Even if the main class instance does not create any graphics itself, it is still added to the display list, ready to contain any graphics created by the program in the future. The main class instance of the first .swf file opened by the Flash runtime plays a special role in ActionScript; it determines certain global environment settings, such as relative-URL resolution and the type of security restrictions applied to external operations.

In honor of its special role, the main-class instance of the first .swf file opened by the Flash runtime is sometimes referred to as the “stage owner.”

Let’s consider an example that shows how the stage owner is created. Suppose we start the standalone version of Flash Player and open a .swf file named GreetingApp.swf, whose main class is GreetingApp. If GreetingApp.swf contains the class GreetingApp only, and GreetingApp creates no graphics, then Flash Player’s display list will contain just two items: the Stage instance and a GreetingApp instance (contained by the Stage instance). Figure 20-3 demonstrates.

Figure 20-3. The display list for GreetingApp.swf

Once an instance of a .swf file’s main class has been added to the Stage instance, a program can add new content to the screen by following these general steps:

1. Create a displayable object (i.e., an instance of any core display class or any class that extends a core display class).
2. Invoke the DisplayObjectContainer class’s instance method addChild() on either the Stage instance or the main-class instance, and pass addChild() the displayable object created in Step 1.

Let’s try out the preceding general steps by creating the GreetingApp class, then adding a rectangle, a circle, and a text field to the display list using addChild(). First, here’s the skeleton of the GreetingApp class:

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

```
public class GreetingApp extends Sprite {
    public function GreetingApp () {
    }
}

Our GreetingApp class will use the Shape and Sprite classes, so it imports the entire flash.display package in which those classes reside. Likewise, GreetingApp will use the TextField class, so it imports flash.text.TextField.

Notice that, by necessity, GreetingApp extends Sprite. GreetingApp must extend either Sprite or MovieClip because it is the program’s main class.

In ActionScript 3.0, a .swf file’s main class must extend either Sprite or MovieClip, or a subclass of one of those classes.

In cases where the main class represents the root timeline of a .fla file, it should extend MovieClip; in all other cases, it should extend Sprite. In our example, GreetingApp extends Sprite because it is not associated with a .fla file. It is intended to be compiled as a standalone ActionScript application.

Now let’s create our rectangle and circle in GreetingApp’s constructor method. We’ll draw both the rectangle and the circle inside a single Shape object. Shape objects (and all graphical objects) are created with the new operator, just like any other kind of object. Here’s the code we use to create a new Shape object:

    new Shape()

Of course, we’ll need to access that object later in order to draw things in it, so let’s assign it to a variable, rectAndCircle:

    var rectAndCircle:Shape = new Shape();

To draw vectors in ActionScript, we use the supporting display class, Graphics. Each Shape object maintains its own Graphics instance in the instance variable graphics. Hence, to draw a rectangle and circle inside our Shape object, we invoke the appropriate methods on rectAndCircle.graphics. Here’s the code:

    // Set line thickness to one pixel
    rectAndCircle.graphics.lineStyle(1);

    // Draw a blue rectangle
    rectAndCircle.graphics.beginFill(0x0000FF, 1);
    rectAndCircle.graphics.drawRect(125, 0, 150, 75);

    // Draw a red circle
    rectAndCircle.graphics.beginFill(0xFF0000, 1);
    rectAndCircle.graphics.drawCircle(50, 100, 50);
For more information on vector drawing in ActionScript 3.0, see Chapter 25.

Vector drawing operations are not limited to the *Shape* class. The *Sprite* class also provides a *Graphics* reference via its instance variable graphics, so we could have created a *Sprite* object to hold the rectangle and circle rather than a *Shape* object. However, because each *Sprite* object requires more memory than each *Shape* object, we’re better off using a *Shape* object when creating vector graphics that do not contain children or require interactivity.

Strictly speaking, if we wanted to incur the lowest possible memory overhead in the *GreetingApp* example, we would draw our shapes directly inside the *GreetingApp* instance (remember *GreetingApp* extends *Sprite*, so it supports vector drawing). The code would look like this:

```actionscript
package {
    import flash.display.*;
    public class GreetingApp extends Sprite {
        public function GreetingApp () {
            graphics.lineStyle(1);
            // Rectangle
            graphics.beginFill(0x0000FF, 1);
            graphics.drawRect(125, 0, 150, 75);

            // Circle
            graphics.beginFill(0xFF0000, 1);
            graphics.drawCircle(50, 100, 50);
        }
    }
}
```

That code successfully draws the rectangle and circle on screen but is less flexible than placing them in a separate *Shape* object. Placing drawings in a *Shape* object allows them to be moved, layered, modified, and removed independent of other graphical content in the application. For example, returning to our earlier approach of drawing in a *Shape* instance (*rectAndCircle*), here’s how we’d move the shapes to a new position:

```actionscript
// Move *rectAndCircle* to the right 125 pixels and down 100 pixels
rectAndCircle.x = 125;
rectAndCircle.y = 100;
```

Notice that at this point in our code, we have a display object, *rectAndCircle*, that has not yet been added to the display list. It’s both legal and common to refer to and manipulate display objects that are not on the display list. Display objects can be added to and removed from the display list arbitrarily throughout the lifespan of a program and can be programmatically manipulated whether they are on or off the
display list. For example, notice that the preceding positioning code occurs before `rectAndCircle` has even been placed on the display list! Each display object maintains its own state regardless of the parent it is attached to—indeed, regardless of whether it is attached to the display list at all. When and if `rectAndCircle` is eventually added to a display container, it is automatically placed at position (125, 100) in that container’s coordinate space. If `rectAndCircle` is then removed from that container and added to a different one, it is positioned at (125, 100) of the new container’s coordinate space.

Each display object carries its characteristics with it when moved from container to container, or even when removed from the display list entirely.

Now the moment we’ve been waiting for. To actually display our rectangle and circle on screen, we invoke `addChild()` on the `GreetingApp` instance within the `GreetingApp` constructor and pass along a reference to the `Shape` instance in `rectAndCircle`.

```javascript
// Display rectAndCircle on screen by adding it to the display list
addChild(rectAndCircle);
```

Flash Player consequently adds `rectAndCircle` to the display list, as a child of the `GreetingApp` instance.

As a `Sprite` subclass, `GreetingApp` is a descendant of `DisplayObjectContainer`, and, thus, inherits the `addChild()` method and the ability to contain children. For a refresher on the display API class hierarchy, refer back to Figure 20-1.

Wow, displaying things on screen is fun! Let’s do it again. Adding the following code to the `GreetingApp` constructor causes the text “Hello world” to appear on screen:

```javascript
// Create a TextField object to contain some text
var greeting_txt:TextField = new TextField();

// Specify the text to display
greeting_txt.text = "Hello world";

// Position the TextField object
greeting_txt.x = 200;
greeting_txt.y = 300;

// Display the text on screen by adding greeting_txt to the display list
addChild(greeting_txt);
```

Once an object has been added to a display container, that container can be accessed via the `DisplayObject` class’s instance variable `parent`. For example, from within the
GreetingApp constructor, the following code is a valid reference to the GreetingApp instance:

greeting_txt.parent

If a display object is not currently on the display list, its parent variable has the value null.

Example 20-1 shows the code for GreetingApp in its entirety.

Example 20-1. Graphical “Hello world”

package {
    import flash.display.*;
    import flash.text.TextField;

    public class GreetingApp extends Sprite {
        public function GreetingApp() {
            // Create the Shape object
            var rectAndCircle:Shape = new Shape();

            // Set line thickness to one pixel
            rectAndCircle.graphics.lineStyle(1);

            // Draw a blue rectangle
            rectAndCircle.graphics.beginFill(0x0000FF, 1);
            rectAndCircle.graphics.drawRect(125, 0, 150, 75);

            // Draw a red circle
            rectAndCircle.graphics.beginFill(0xFF0000, 1);
            rectAndCircle.graphics.drawCircle(50, 100, 50);

            // Move the shape to the right 125 pixels and down 100 pixels
            rectAndCircle.x = 125;
            rectAndCircle.y = 100;

            // Show rectAndCircle on screen by adding it to the display list
            addChild(rectAndCircle);

            // Create a TextField object to contain some text
            var greeting_txt:TextField = new TextField();

            // Specify the text to display
            greeting_txt.text = "Hello world";

            // Position the text
            greeting_txt.x = 200;
            greeting_txt.y = 300;

            // Show the text on screen by adding greeting_txt to the display list
            addChild(greeting_txt);
        }
    }
}
Figure 20-4 shows the graphical results of the code in Example 20-1. As in the previous two figures, on-screen graphics are depicted on the left, with the corresponding Flash Player display list hierarchy shown on the right.

Containers and Depths

In the previous section, we gave GreetingApp two display children (rectAndCircle and greeting_txt). On screen, those two children were placed in such a way that they did not visually overlap. If they had overlapped, one would have obscured the other, based on the depths of the two objects. A display object’s depth is an integer value that determines how that object overlaps other objects in the same display object container. When two display objects overlap, the one with the greater depth position (the “higher” of the two) obscures the other (the “lower” of the two). All display objects in a container, hence, can be thought of as residing in a visual stacking order akin to a deck of playing cards, counted into a pile starting at zero. The lowest object in the stacking order has a depth position of 0, and the highest object has a depth position equal to the number of child objects in the display object container, minus one (metaphorically, the lowest card in the deck has a depth position of 0, and the highest card has a depth position equal to the number of cards in the deck, minus one).

ActionScript 2.0’s depth-management API allowed “unoccupied” depths. For example, in a container with only two objects, one object might have a depth of 0 and the other a depth of 40, leaving depths 1 through 39 unoccupied. In ActionScript 3.0’s depth-management API, unoccupied depths are no longer allowed or necessary.

Display objects added to a container using addChild() are assigned depth positions automatically. Given an empty container, the first child added via addChild() is placed at depth 0, the second is placed at depth 1, the third is placed at depth 2, and
so on. Hence, the object most recently added via `addChild()` always appears visually on top of all other children.

As an example, let’s continue with the `GreetingApp` program from the previous section. This time we’ll draw the circle and rectangle in their own separate `Shape` instances so they can be stacked independently. We’ll also adjust the positions of the circle, rectangle, and text so that they overlap. Here’s the revised code (this code and other samples in this section are excerpted from `GreetingApp`’s constructor method):

```actionscript
// The rectangle
var rect:Shape = new Shape();
rect.graphics.lineStyle(1);
rect.graphics.beginFill(0x0000FF, 1);
rect.graphics.drawRect(0, 0, 75, 50);

// The circle
var circle:Shape = new Shape();
circle.graphics.lineStyle(1);
circle.graphics.beginFill(0xFF0000, 1);
circle.graphics.drawCircle(0, 0, 25);
circle.x = 75;
circle.y = 35;

// The text message
var greeting_txt:TextField = new TextField();
greeting_txt.text = "Hello world";
greeting_txt.x = 60;
greeting_txt.y = 25;
```

Now let’s try adding the rectangle, circle, and text as `GreetingApp` children, in different sequences. This code adds the rectangle, then the circle, then the text:

```actionscript
addChild(rect);          // Depth 0
addChild(circle);        // Depth 1
addChild(greeting_txt);  // Depth 2
```

As shown in Figure 20-5, the rectangle was added first, so it appears underneath the circle and the text; the circle was added next, so it appears on top of the rectangle but underneath the text; the text was added last, so it appears on top of both the circle and the rectangle.

![Figure 20-5. Rectangle, circle, text](image)

The following code changes the sequence, adding the circle first, then the rectangle, then the text. Figure 20-6 shows the result. Notice that simply changing the sequence in which the objects are added changes the resulting display.
addChild(circle); // Depth 0
addChild(rect); // Depth 1
addChild(greeting_txt); // Depth 2

Here’s one more example. The following code adds the text first, then the circle, then the rectangle. Figure 20-7 shows the result.

addChild(greeting_txt); // Depth 0
addChild(circle); // Depth 1
addChild(rect); // Depth 2

To retrieve the depth position of any object in a display object container, we use the `DisplayObjectContainer` class’s instance method `getChildIndex()`:

```
trace(getChildIndex(rect)); // Displays: 2
```

To add a new object at a specific depth position, we use the `DisplayObjectContainer` class’s instance method `addChildAt()` (notice: `addChildAt()` not `addChild()`). The `addChildAt()` method takes the following form:

```
theContainer.addChildAt(theDisplayObject, depthPosition)
```

The `depthPosition` must be an integer between 0 and `theContainer.numChildren`, inclusive.

If the specified `depthPosition` is already occupied by an existing child, then `theDisplayObject` is placed behind that existing child (i.e., the depth positions of all display objects on or above that depth increases by one to make room for the new child).

```
Repeat this `addChildAt()` mnemonic to yourself: “If the depth is occupied, the new child goes behind.”
```

To add a new object above all existing children, we use:

```
theContainer.addChildAt(theDisplayObject, theContainer.numChildren)
```
which is synonymous with the following:

```javascript
theContainer.addChild(theDisplayObject)
```

Typically, `addChildAt()` is used in combination with the `DisplayObjectContainer` class’s instance method `getChildIndex()` to add an object below an existing child in a given container. Here’s the general format:

```javascript
theContainer.addChildAt(newChild, theContainer.getChildAt(existingChild))
```

Let’s try it out by adding a new triangle behind the circle in `GreetingApp` as it existed in its most recent incarnation, shown in Figure 20-7.

Here’s the code that creates the triangle:

```javascript
var triangle:Shape = new Shape();
triangle.graphics.lineStyle(1);
triangle.graphics.beginFill(0x00FF00, 1);
triangle.graphics.moveTo(25, 0);
triangle.graphics.lineTo(50, 25);
triangle.graphics.lineTo(0, 25);
triangle.graphics.lineTo(25, 0);
triangle.graphics.endFill();
triangle.x = 25;
triangle.y = 10;
```

And here’s the code that makes `triangle` a new child of `GreetingApp`, beneath the existing object, `circle` (notice that both `addChildAt()` and `getChildIndex()` are implicitly invoked on the current `GreetingApp` object). Figure 20-8 shows the results.

```javascript
addChildAt(triangle, getChildIndex(circle));
```

![Figure 20-8. New triangle child](image-url)

As we learned recently, when a new object is added at a depth position occupied by an existing child, the depth positions of the existing child and of all children above it are incremented by 1. The new object then adopts the depth position that was vacated by the existing child. For example, prior to the addition of `triangle`, the depths of `GreetingApp`’s children were:

- `greeting_txt`: 0
- `circle`: 1
- `rect`: 2

Upon adding `triangle`, `circle`’s depth position changes from 1 to 2, `rect`’s depth position changes from 2 to 3, and `triangle` takes depth 1 (`circle`’s former depth). Meanwhile, `greeting_txt`’s depth position is unaffected because it was below
circle’s depth from the beginning. Here are the revised depths after the addition of
triangle:

<table>
<thead>
<tr>
<th>Object</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>greeting_txt</td>
<td>0</td>
</tr>
<tr>
<td>triangle</td>
<td>1</td>
</tr>
<tr>
<td>circle</td>
<td>2</td>
</tr>
<tr>
<td>rect</td>
<td>3</td>
</tr>
</tbody>
</table>

To change the depth of an existing child, we can swap that child’s depth position
with another existing child via the DisplayObjectContainer class’s instance methods
swapChildren() or swapChildrenAt(). Or, we can simply set that child’s depth
directly using the DisplayObjectContainer class’s instance method setChildIndex().

The swapChildren() method takes the following form:

```javascript
theContainer.swapChildren(existingChild1, existingChild2);
```

where existingChild1 and existingChild2 are both children of theContainer. The
swapChildren() method exchanges the depths of existingChild1 and existingChild2.
In natural English, the preceding code means, “put existingChild1 at the depth cur-
rently occupied by existingChild2, and put existingChild2 at the depth currently
occupied by existingChild1.”

The swapChildrenAt() method takes the following form:

```javascript
theContainer.swapChildrenAt(existingDepth1, existingDepth2);
```

where existingDepth1 and existingDepth2 are both depths occupied by children of
theContainer. The swapChildrenAt() method exchanges the depths of the children at
existingDepth1 and existingDepth2. In natural English, the preceding code means,
“put the child currently at existingDepth1 at existingDepth2, and put the child cur-
rently at existingDepth2 at existingDepth1.”

The setChildIndex() method takes the following form:

```javascript
theContainer.setChildIndex(existingChild, newDepthPosition);
```

where existingChild is a child of theContainer. The newDepthPosition must be a
depth position presently occupied by a child object of theContainer. That is,
setChildIndex() can only rearrange the positions of existing child objects; it cannot
introduce new depth positions. The newDepthPosition parameter of setChildIndex()
is typically deduced by invoking getChildIndex() on an existing child, as in:

```javascript
theContainer.setChildIndex(existingChild1,
  theContainer.getChildIndex(existingChild2));
```

which means, “put existingChild1 at the depth currently occupied by
existingChild2.”

Note that when an object’s depth is increased to a new position via setChildIndex()
(i.e., the object is moved higher), the depth of all objects between the old position
and the new position is decreased by 1, thus filling the vacant position left by the
moved object. Consequently, the moved object appears in front of the object for-
merly at the new position. For example, continuing with the latest version of
GreetingApp (as shown previously in Figure 20-8), let’s change greeting_txt’s depth position from 0 to 2. Prior to executing the following code, depth position 2 is held by circle.

```
setChildIndex(greeting_txt, getChildIndex(circle));
```

When greeting_txt moves to depth position 2, the depth positions of circle and triangle are reduced to 1 and 0, respectively, so greeting_txt appears in front of them both. See Figure 20-9.

![Figure 20-9. Moving the text higher](image)

By contrast, when an object’s depth is decreased to a new position via `setChildIndex()` (i.e., the object is moved lower), the depth position of all objects at or above the new position is increased by 1, thus making room for the new object. Consequently, the moved object appears behind the object formerly at the new position (exactly as if the object had been added with `addChildAt()`). Notice the important difference between moving an object to a higher depth versus moving it to a lower depth.

An object moved to a higher depth appears in front of the object at the target position, but an object moved lower appears behind the object at the target position.

For example, continuing from Figure 20-9, let’s change rect’s depth position from 3 to 1 (where 1 is the depth currently held by circle):

```
setChildIndex(rect, getChildIndex(circle));
```

When rect moves to depth position 1, the depth positions of circle and greeting_txt are increased to 2 and 3, respectively, so rect appears behind them both (see Figure 20-10).

![Figure 20-10. Moving the rectangle lower](image)

To move an object to the top of all objects in a given container, use:

```
theContainer.setChildIndex(existingChild, theContainer.numChildren-1)
```
For example, the following code moves the triangle to the top of `GreetingApp`'s children (the following code occurs within the `GreetingApp` class, so `theContainer` is omitted and implicitly resolves to this, the current object):

```javascript
setChildIndex(triangle, numChildren-1);
```

Figure 20-11 shows the results.

![Figure 20-11. Triangle moved to front](image)

The `setChildIndex()` method is easy to understand if you think of a `DisplayObjectContainer`'s children as being modeled after a deck of cards, as discussed earlier. If you move a card from the bottom of the deck to the top, the other cards all move down (i.e., the card that used to be just above the bottom card is now, itself, the new bottom card). If you move a card from the top of the deck to the bottom, the other cards all move up (i.e., the card that used to be the bottom card is now one above the new bottom card).

### Removing Assets from Containers

To remove an object from a display object container, we use the `DisplayObjectContainer` class's instance method `removeChild()`, which takes the following form:

```javascript
theContainer.removeChild(existingChild)
```

where `theContainer` is a container that currently contains `existingChild`. For example, to remove the triangle from `GreetingApp` we’d use:

```javascript
removeChild(triangle);
```

Alternatively, we can remove a child based on its depth using `removeChildAt()`, which takes the following form:

```javascript
theContainer.removeChildAt(depth)
```

After `removeChild()` or `removeChildAt()` runs, the removed child's parent variable is set to `null` because the removed child has no container. If the removed child was on the display list before the call to `removeChild()` or `removeChildAt()`, it is removed from the display list. If the removed child was visible on screen before the call to `removeChild()` or `removeChildAt()`, it is also removed from the screen. If the removed child is, itself, a `DisplayObjectContainer` with its own children, those children are also removed from the screen.
Removing Assets from Memory

It’s important to note that the `removeChild()` and `removeChildAt()` methods discussed in the previous section do not necessarily cause the removed object to be purged from memory; they only remove the object from the parent `DisplayObjectContainer` object’s display hierarchy. If the removed object is referenced by a variable or array element, it continues to exist and can be re-added to another container at a later time. For example, consider the following code, which creates a `Shape` object, assigns it to the variable `rect`, and then adds it to `parent`’s display hierarchy:

```javascript
var rect:Shape = new Shape();
rect.graphics.lineStyle(1);
rect.graphics.beginFill(0x0000FF, 1);
rect.graphics.drawRect(0, 0, 75, 50);
parent.addChild(rect);
```

If we now use `removeChild()` to remove the `Shape` object from `parent`, `rect` continues to refer to the `Shape` object:

```
parent.removeChild(rect);
trace(rect);  // Displays: [object Shape]
```

As long as the `rect` variable exists, we can use it to re-add the `Shape` object to `parent`’s display hierarchy, as follows:

```
parent.addChild(rect);
```

To completely remove a display object from a program, we must both remove it from the screen using `removeChild()` and also remove all references to it. To remove all references to the object, we must manually remove it from every array that contains it and assign `null` (or some other value) to every variable that references it. Once all references to the object have been removed, the object becomes eligible for garbage collection and will eventually be removed from memory by ActionScript’s garbage collector.

However, as discussed in Chapter 14, even after all references to an object have been removed, that object continues to be active until the garbage collector deletes it from memory. For example, if the object has registered listeners for the `Event.ENTER_FRAME` event, that event will still trigger code execution. Likewise, if the object has started timers using `setInterval()` or the `Timer` class, those timers will still trigger code execution. Similarly, if the object is a `MovieClip` instance that is playing, its playhead will continue to advance, causing any frame scripts to execute.

While an object is waiting to be garbage collected, event listeners, timers, and frame scripts can cause unnecessary code execution, resulting in memory waste or undesired side effects.
To avoid unnecessary code execution when removing a display object from a program, be sure that, before releasing all references to the object, you completely disable it. For more important details on disabling objects, see Chapter 14.

Always disable display objects before discarding them.

**Removing All Children**

ActionScript does not provide a direct method for removing all of an object’s children. Hence, to remove every display child from a given object, we must use a `while` loop or a `for` loop. For example, the following code uses a `while` loop to remove all children of `theParent` from the bottom up. First, the child at depth 0 is removed, then the depth of all children is reduced by 1, then the new child at depth 0 is removed, and the process repeats until there are no children left.

```java
// Remove all children of theParent
while (theParent.numChildren > 0) {
    theParent.removeChildAt(0);
}
```

The following code also removes all children of `theParent`, but from the top down. It should be avoided because it is slower than the preceding approach of removing children from the bottom up.

```java
while (theParent.numChildren > 0) {
    theParent.removeChildAt(theParent.numChildren-1);
}
```

The following code removes all children, from the bottom up, using a `for` loop instead of a `while` loop:

```java
for (;numChildren > 0;) {
    theParent.removeChildAt(0);
}
```

If you must remove children from the top down (perhaps because you need to process them in that order before removal), be careful never to use a loop that increments its counter instead of decrementing it. For example, never use code like this:

```java
// WARNING: PROBLEM CODE! DO NOT USE!
for (var i:int = 0; i < theParent.numChildren; i++) {
    theParent.removeChildAt(i);
}
```

What’s wrong with the preceding `for` loop? Imagine `theParent` has three children: A, B, and C, positioned at depths 0, 1, and 2, respectively:

<table>
<thead>
<tr>
<th>Children</th>
<th>Depths</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>
When the loop runs the first time, \( i \) is 0, so \( A \) is removed. When \( A \) is removed, \( B \) and \( C \)'s depth is automatically reduced by 1, so \( B \)'s depth is now 0 and \( C \)'s depth is now 1:

<table>
<thead>
<tr>
<th>Children</th>
<th>Depths</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B )</td>
<td>0</td>
</tr>
<tr>
<td>( C )</td>
<td>1</td>
</tr>
</tbody>
</table>

When the loop runs for the second time, \( i \) is 1, so \( C \) is removed. With \( C \) removed, \( \text{theParent}.\text{numChildren} \) becomes 1, so the loop ends because \( i \) is no longer less than \( \text{theParent}.\text{numChildren} \). But \( B \) was never removed (sneaky devil)!

**Reparenting Assets**

In ActionScript 3.0, it's perfectly legal and common to remove a child from one \( \text{DisplayObjectContainer} \) instance and move it to another. In fact, the mere act of adding an object to a container automatically removes that object from any container it is already in.

To demonstrate, Example 20-2 presents a simple application, \( \text{WordHighlighter} \), in which a \( \text{Shape} \) object (assigned to the variable \( \text{bgRect} \)) is moved between two \( \text{Sprite} \) instances (assigned to the variables \( \text{word1} \) and \( \text{word2} \)). The \( \text{Sprite} \) instances contain \( \text{TextField} \) instances (assigned to the variables \( \text{text1} \) and \( \text{text2} \)) that display the words Products and Services. The \( \text{Shape} \) is a rounded rectangle that serves to highlight the word currently under the mouse pointer, as shown in Figure 20-12. When the mouse hovers over one of the \( \text{TextField} \) instances, the \( \text{Shape} \) object is moved to the \( \text{Sprite} \) containing that \( \text{TextField} \).

![Figure 20-12. Moving an object between containers](image)

We haven't yet covered the mouse-event handling techniques used in Example 20-2. For information on handling input events, see Chapter 22.

**Example 20-2. Moving an object between containers**

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

    public class WordHighlighter extends Sprite {
        // The first word
        private var word1:Sprite;
        private var text1:TextField;

        // The second word
        private var word2:Sprite;
        private var text2:TextField;
    }
}
```
Example 20-2. Moving an object between containers (continued)

    // The highlight shape
    private var bgRect:Shape;
    public function WordHighlighter () {
        // Create the first TextField and Sprite
        word1 = new Sprite();
        text1 = new TextField();
        text1.text = "Products";
        text1.selectable = false;
        text1.autoSize = TextFieldAutoSize.LEFT;
        word1.addChild(text1)
        text1.addEventListener(MouseEvent.MOUSE_OVER, mouseOverListener);

        // Create the second TextField and Sprite
        word2 = new Sprite();
        text2 = new TextField();
        text2.text = "Services";
        text2.selectable = false;
        text2.autoSize = TextFieldAutoSize.LEFT;
        word2.x = 75;
        word2.addChild(text2)
        text2.addEventListener(MouseEvent.MOUSE_OVER, mouseOverListener);

        // Add the Sprite instances to WordHighlighter's display hierarchy
        addChild(word1);
        addChild(word2);

        // Create the Shape (a rounded rectangle)
        bgRect = new Shape();
        bgRect.graphics.lineStyle(1);
        bgRect.graphics.beginFill(0xCCCCCC, 1);
        bgRect.graphics.drawRoundRect(0, 0, 60, 15, 8);
    }

    // Invoked when the mouse pointer moves over a text field.
    private function mouseOverListener (e:MouseEvent):void {
        // If the TextField's parent Sprite does not already contain
        // the shape, then move it there. DisplayObjectContainer.contains()
        // returns true if the specified object is a descendant
        // of the container.
        if (!e.target.parent.contains(bgRect)) {
            e.target.parent.addChildAt(bgRect, 0);
        }
    }

As it stands, the code in Example 20-2 always leaves one of the text fields highlighted. To remove the highlight when the mouse moves away from both text fields, we would first register both text fields to receive the MouseEvent.MOUSE_OUT event:

    text1.addEventListener(MouseEvent.MOUSE_OUT, mouseOutListener);
    text2.addEventListener(MouseEvent.MOUSE_OUT, mouseOutListener);
Then, we would implement code to remove the rectangle in response to MouseEvent.

```
private function mouseOutListener (e:MouseEvent):void {
    // If the highlight is present...
    if (e.target.parent.contains(bgRect)) {
        // ...remove it
        e.target.parent.removeChild(bgRect);
    }
}
```

**Traversing Objects in a Display Hierarchy**

To traverse objects in a display hierarchy means to systematically access some or all of a container’s child objects, typically to manipulate them in some way.

To access the *direct* children of a container (but not grandchildren or any other descendant children), we use a loop statement. The loop iterates over each depth position in the container. Within the loop body, we access each child according to its depth using the `DisplayObjectContainer` class’s instance method `getChildAt()`. The following code shows the general technique; it displays the string value of all objects contained by `theContainer`:

```
for (var i:int=0; i < theContainer.numChildren; i++) {
    trace(theContainer.getChildAt(i).toString());
}
```

Example 20-3 shows a more concrete, if whimsical, application of display object children traversal. It creates 20 `Shape` instances containing rectangles and then uses the preceding traversal technique to rotate those instances when the mouse is clicked. The traversal code is shown in bold. (In upcoming chapters, we’ll study both the vector-drawing techniques and mouse-event-handling techniques used in the example.)

**Example 20-3. Rotating rectangles**

```actionscript
class RotatingRectangles extends Sprite {

    public function RotatingRectangles () {
        // Create 20 rectangles
        var rects:Array = new Array();
        for (var i:int = 0; i < 20; i++) {
            rects[i] = new Shape();
            rects[i].graphics.lineStyle(1);
            rects[i].graphics.beginFill(Math.floor(Math.random()*0xFFFFFF), 1);
            rects[i].graphics.drawRect(0, 0, 100, 50);
            rects[i].x = Math.floor(Math.random()*500);
            rects[i].y = Math.floor(Math.random()*400);
            addChild(rects[i]);
        }
    }
}
```
To access not just the direct children of a container, but all of its descendants, we combine the preceding for loop with a recursive function. Example 20-4 shows the general approach.

Example 20-4. Recursive display list tree traversal

```actionscript
public function processChildren (container:DisplayObjectContainer):void {
    for (var i:int = 0; i < container.numChildren; i++) {
        // Process the child here. For example, the following line
        // prints this child's string value as debugging output.
        var thisChild:DisplayObject = container.getChildAt(i);
        trace(thisChild.toString());

        // If this child is, itself, a container, then process its children.
        if (thisChild is DisplayObjectContainer) {
            processChildren(DisplayObjectContainer(thisChild));
        }
    }
}
```

The following function, rotateChildren(), applies the generalized code from Example 20-4. It randomly rotates all the descendants of a specified container (not just the children). However, notice the minor change in the approach from Example 20-4: rotateChildren() only rotates noncontainer children.

```actionscript
public function rotateChildren (container:DisplayObjectContainer):void {
    for (var i:int = 0; i < container.numChildren; i++) {
        var thisChild:DisplayObject = container.getChildAt(i);
        if (thisChild is DisplayObjectContainer) {
            rotateChildren(DisplayObjectContainer(thisChild));
        } else {
            thisChild.rotation = Math.floor(Math.random()*360);
        }
    }
}
```
Manipulating Objects in Containers Collectively

In the earlier section “Display API Overview,” we learned that child objects automatically move, rotate, and transform when their ancestors are moved, rotated, and transformed. We can use this feature to perform collective visual modifications to groups of objects. To learn how, let’s create two rectangular Shape instances in a Sprite instance:

```javascript
// Create two rectangles
var rect1:Shape = new Shape();
rect1.graphics.lineStyle(1);
rect1.graphics.beginFill(0x0000FF, 1);
rect1.graphics.drawRect(0, 0, 75, 50);

var rect2:Shape = new Shape();
rect2.graphics.lineStyle(1);
rect2.graphics.beginFill(0xFF0000, 1);
rect2.graphics.drawRect(0, 0, 75, 50);
rect2.x = 50;
rect2.y = 75;

// Create the container
var group:Sprite = new Sprite();

// Add the rectangles to the container
group.addChild(rect1);
group.addChild(rect2);

// Add the container to the main application
someMainApp.addChild(group);
```

Figure 20-13 shows the result.

![Figure 20-13. Two rectangles in a container](image)

Now let’s move, scale, and rotate the container, as follows:

```javascript
group.x = 40;
group.scaleY = .15;
group.rotation = 15;
```

The modifications affect the child Shape instances, as shown in Figure 20-14.
A container’s transformations also affect children added after the transformations are applied. For example, if we now add a third rectangular Shape to group, that Shape is moved, scaled, and rotated according to group’s existing transformations:

```
// Create a third rectangle
var rect3:Shape = new Shape();
rect3.graphics.lineStyle(1);
rect3.graphics.beginFill(0x00FF00, 1);
rect3.graphics.drawRect(0, 0, 75, 50);
rect3.x = 25;
rect3.y = 35;
group.addChild(rect3);
```

Figure 20-15 shows the result.

At any time, we can remove or change the container’s transformation, and all children will be affected. For example, the following code restores the container to its original state:

```
group.scaleY = 1;
group.x = 0;
group.rotation = 0;
```

Figure 20-16 shows the result. Notice that the third rectangle now appears in its true dimensions and position.

Color and coordinate transformations made to a container via the DisplayObject class’s instance variable transform are also inherited by its descendants. For example, the following code applies a black color transformation to group, causing all three rectangles to be colored solid black.
import flash.geom.ColorTransform;
var blackTransform:ColorTransform = new ColorTransform();
blackTransform.color = 0x000000;
group.transform.colorTransform = blackTransform;

For complete details on the types of color and coordinate transformations available in ActionScript, see flash.geom.Transform in Adobe's ActionScript Language Reference.

Transformations made to nested containers are compounded. For example, the following code places a rectangle in a Sprite that is nested within another Sprite. Both Sprite instances are rotated 45 degrees. As a result, the rectangle appears rotated on screen by 90 degrees (45 + 45).

// Create a rectangle
var rect1:Shape = new Shape();
rect1.graphics.lineStyle(1);
rect1.graphics.beginFill(0x0000FF, 1);
rect1.graphics.drawRect(0, 0, 75, 50);
var outerGroup:Sprite = new Sprite();
var innerGroup:Sprite = new Sprite();
innerGroup.addChild(rect1);
outerGroup.addChild(innerGroup);
innerGroup.rotation = 45;
outerGroup.rotation = 45;

**Descendant Access to a .swf File’s Main Class Instance**

In ActionScript 3.0, the display descendants of a .swf file’s main class instance can retrieve a reference to that instance via the DisplayObject class’s instance variable root. For example, consider Example 20-5, which shows a .swf file’s main class, App. When the code runs, ActionScript automatically creates an App instance and runs its constructor. Within the constructor, two App instance descendants (a Sprite object and a Shape object) use root to access the App instance.
When an object is on the display list but is not a descendant of a .swf file’s main class instance, its root variable returns a reference to the Stage instance. For example, the following code modifies the App class from Example 20-5 so that the Sprite object and its child Shape object are added directly to the Stage instance. Because the Sprite and Shape objects are not descendants of a .swf file’s main class instance, their root variables refer to the Stage instance.

```actionscript
package {
    import flash.display.*;
    import flash.geom.*;

    public class App extends Sprite {
        public function App () {
            var rect:Shape = new Shape();
            rect.graphics.lineStyle(1);
            rect.graphics.beginFill(0x0000FF, 1);
            rect.graphics.drawRect(0, 0, 75, 50);
            var sprite:Sprite = new Sprite();
            sprite.addChild(rect);
            addChild(sprite);

            // Use DisplayObject.root to access this App instance
            trace(rect.root);    // Displays: [object App]
            trace(sprite.root);  // Displays: [object App]
        }
    }
}
```

Example 20-5. Descendant access to a .swf file’s main class instance

When an object is on the display list but is not a descendant of a .swf file’s main class instance, its root variable returns a reference to the Stage instance. For example, the following code modifies the App class from Example 20-5 so that the Sprite object and its child Shape object are added directly to the Stage instance. Because the Sprite and Shape objects are not descendants of a .swf file’s main class instance, their root variables refer to the Stage instance.

```actionscript
package {
    import flash.display.*;
    import flash.geom.*;

    public class App extends Sprite {
        public function App () {
            var rect:Shape = new Shape();
            rect.graphics.lineStyle(1);
            rect.graphics.beginFill(0x0000FF, 1);
            rect.graphics.drawRect(0, 0, 75, 50);
            var sprite:Sprite = new Sprite();
            sprite.addChild(rect);
            addChild(sprite);

            // Add child to Stage instance, not this App instance
            stage.addChild(sprite);

            trace(rect.root);    // Displays: [object Stage]
            trace(sprite.root);  // Displays: [object Stage]
        }
    }
}
```
For objects that are on the display list but are not descendants of a .swf file’s main-class instance, the DisplayObject class’s instance variable root is synonymous with its instance variable stage.

In the first .swf file opened by a Flash runtime, the root variable of display objects that are not on the display list has the value null.

In .swf files loaded by other .swf files, the root variable is set as follows:

- For display objects that are display descendants of the main class instance, the root variable refers to that instance, even if the main class instance is not on the display list.
- For display objects that are not display descendants of the main class instance and are not on the display list, the root variable has the value null.

The rebirth of _root

In ActionScript 2.0 and older versions of the language, the global _root variable referred to the top-level movie clip of the current _level. Prior to ActionScript 3.0, conventional wisdom held that _root should be avoided because its meaning was volatile (the object to which it referred changed when loading a .swf file into a movie clip).

In ActionScript 3.0, the DisplayObject class’s instance variable root replaces the global _root variable. DisplayObject’s root variable does not suffer from its predecessor’s volatility and is considered a clean, safe member of the display API.

Longtime ActionScript programmers who are used to avoiding the legacy _root variable should feel neither fear nor guilt when using the DisplayObject class’s instance variable root in ActionScript 3.0.

Whither _level0?

In ActionScript 1.0 and 2.0, the loadMovieNum() function was used to stack external .swf files on independent Flash Player levels. Each level was referred to using the format: _leveln, where n indicated the level’s numeric order in the level stack. As of ActionScript 3.0, the concept of levels has been completely removed from the Flash runtime API.

The closest analogue to levels in ActionScript 3.0 is the Stage instance’s children. However, whereas in ActionScript 1.0 and 2.0, external .swf files could be loaded directly onto a _level, in ActionScript 3.0, external .swf files cannot be loaded directly into the Stage instance’s child list. Instead, to add an external .swf file to the Stage instance’s child list, we must first load that .swf file via a Loader object and then move it to the Stage instance via stage.addChild(), as follows:
Furthermore, it is no longer possible to remove all content in Flash Player by unloading _level0. Code such as the following is no longer valid:

```javascript
// Clear all content in Flash Player. Deprecated in ActionScript 3.0.
unloadMovieNum(0);
```

The closest ActionScript 3.0 replacement for `unloadMovieNum(0)` is:

```javascript
stage.removeChildAt(0);
```

Using `stage.removeChildAt(0)` removes the `Stage` instance’s first child from the display list but does not necessarily remove it from the program. If the program maintains other references to the child, the child will continue to exist, ready to be re-added to some other container. As shown in the earlier section “Removing Assets from Memory,” to completely remove a display object from a program, we must both remove it from its container and remove all references to it. Furthermore, invoking `stage.removeChildAt(0)` affects the `Stage` instance’s first child only; other children are not removed from the display list (contrast this with ActionScript 1.0 and 2.0, where invoking `unloadMovieNum(0)` removed all content from all _levels_).

To remove all children of the `Stage` instance, we use the following code within the object that currently resides at depth 0 of the `Stage` instance:

```javascript
while (stage.numChildren > 0) {
    stage.removeChildAt(stage.numChildren-1);
    // When the last child is removed, stage is set to null, so quit
    if (stage == null) {
        break;
    }
}
```

Likewise, the following legacy code—which clears Flash Player of all content and then places `newContent.swf` on _level0—is no longer valid:

```javascript
loadMovieNum("newContent.swf", 0);
```

And there is no ActionScript 3.0 equivalent. However, future versions of ActionScript might re-introduce the ability to clear a Flash runtime of all content, replacing it with a new external `.swf` file.

## Containment Events

Earlier we learned how to use the `addChild()` and `addChildAt()` methods to add a new display child to a `DisplayObjectContainer` object. Recall the general code:

```javascript
// The addChild() method
someContainer.addChild(newChild)

// The addChildAt() method
someContainer.addChild(newChild, depth)
```
We also learned that existing child display objects can be removed from a `DisplayObjectContainer` object via the `removeChild()` and `removeChildAt()` methods. Again, recall the following general code:

```actionscript
// The removeChild() method
someContainer.removeChild(childToRemove)
// The removeChildAt() method
someContainer.removeChildAt(depthOfChildToRemove)
```

Finally, we learned that an existing child display object can be removed from a container by moving that child to another container via either `addChild()` and `addChildAt()`. Here’s the code:

```actionscript
// Add child to someContainer
someContainer.addChild(child)

// Remove child from someContainer by moving it to someOtherContainer
someOtherContainer.addChild(child)
```

Each of these child additions and removals is accompanied by a built-in Flash runtime event—either `Event.ADDED` or `Event.REMOVED`. The following three sections explore how these two events are used in display programming.

The following sections require a good understanding of ActionScript’s hierarchical event dispatch system, as discussed in Chapter 21. If you are not yet thoroughly familiar with hierarchical event dispatch, read Chapter 21 before continuing with the following sections.

### The Event.ADDED and Event.REMOVED Events

When a new child display object is added to a `DisplayObjectContainer` object, ActionScript dispatches an `Event.ADDED` event targeted at the new child. Likewise, when an existing child display object is removed from a `DisplayObjectContainer` object, ActionScript dispatches an `Event.REMOVED` event targeted at the removed child.

As discussed in Chapter 21, when an event dispatch targets an object in a display hierarchy, that object and all of its ancestors are notified of the event. Hence, when the `Event.ADDED` event occurs, the added child, its new parent container, and all ancestors of that container are notified that the child was added. Likewise, when the `Event.REMOVED` event occurs, the removed child and its old parent container and all ancestors of that container are notified that the child is about to be removed. Therefore the `Event.ADDED` and `Event.REMOVED` events can be used in two different ways:

- A `DisplayObjectContainer` instance can use the `Event.ADDED` and `Event.REMOVED` events to detect when it has gained or lost a display descendant.
- A `DisplayObject` instance can use the `Event.ADDED` and `Event.REMOVED` events to detect when it has been added to or removed from a parent container.
Let’s take a look at some generalized code that demonstrates the preceding scenarios, starting with a container detecting a new descendant.

We’ll start by creating two Sprite objects: one to act as a container and the other as a child:

```javascript
var container:Sprite = new Sprite();
var child:Sprite = new Sprite();
```

Next we create a listener method, `addEventListener()`, to register with `container` for Event.ADDED events:

```javascript
private function addedListener (e:Event):void {
    trace("Added was triggered");
}
```

Then we register `addEventListener()` with `container`:

```javascript
container.addEventListener(Event.ADDED, addedListener);
```

Finally, we add `child` to `container`:

```javascript
container.addChild(child);
```

When the preceding code runs, the Flash runtime dispatches an Event.ADDED event targeted at `child`. As a result, because `container` is a display ancestor of `child`, the `addedListener()` function that we registered with `container` is triggered during the event’s bubbling phase (for more on bubbling, see Chapter 21).

When the Event.ADDED event triggers an event listener during the capture phase or the bubbling phase, we know that the object with which the listener registered has a new display descendant.

Now let’s add a new child to `child`, making `container` a proud grandparent:

```javascript
var grandchild:Sprite = new Sprite();
child.addChild(grandchild);
```

When the preceding code runs, the Flash runtime again dispatches an Event.ADDED event, this time targeted at `grandchild`, and `addedListener()` is again triggered during the bubbling phase. Because the listener is triggered during the bubbling phase, we know that `container` has a new descendant, but we’re not sure whether that descendant is a direct child of `container`. To determine whether the new descendant is a direct child of `container`, we check if the `child`’s `parent` variable is equal to the `container` object, as follows:

```javascript
private function addedListener (e:Event):void {
    // Remember that Event.currentTarget refers to the object
    // that registered the currently executing listener—in
    // this case, container. Remember also that Event.target
    // refers to the event target, in this case grandchild.
    if (DisplayObject(e.target.parent) == e.currentTarget) {
        trace("A direct child was added");
    }
}
```
Continuing with our example, let’s make container feel like a kid again by adding it (and, by extension, its two descendants) to the Stage instance:

```javascript
stage.addChild(container);
```

When the preceding code runs, the Flash runtime dispatches an Event.ADDED event targeted at container. Once again, addedListener() is triggered—this time during the target phase, not the bubbling phase. Because the listener is triggered during the target phase, we know that container, itself, has been added to a parent container.

To distinguish between container gaining a new descendant and container, itself, being added to a parent container, we examine the current event phase, as follows:

```javascript
private function addedListener (e:Event):void {
    // If this listener was triggered during the capture or bubbling phases...
    if (e.eventPhase != EventPhase.AT_TARGET) {
        // ...then container has a new descendant
        trace("new descendant: " + e.target);
    } else {
        // ...otherwise, container was added to a new parent
        trace("new parent: " + DisplayObject(e.target).parent);
    }
}
```

Now let’s turn to the Event.REMOVED event. It works just like the Event.ADDED event, but is triggered by object removals rather than additions:

The following code registers an Event.REMOVED listener, named removedListener(), with container for the Event.REMOVED event:

```javascript
container.addEventListener(Event.REMOVED, removedListener);
```

Now let’s remove a descendant from the container object:

```javascript
child.removeChild(grandchild)
```

When the preceding code runs, the Flash runtime dispatches an Event.REMOVED event targeted at grandchild, and removedListener() is triggered during the bubbling phase.

Next, the following code removes container, itself, from the Stage instance:

```javascript
stage.removeChild(container)
```
When the preceding code runs, the Flash runtime dispatches an `Event.REMOVED` event targeted at `container`, and `removedListener()` is triggered during the target phase.

Just as with `addedListener()`, within `removedListener()` we can distinguish between `container` losing a descendant and `container`, itself, being removed from its parent `container` by examining the current event phase, as follows:

```actionscript
private function removedListener (e:Event):void {
    // If this listener was triggered during the capture or bubbling phases...
    if (e.eventPhase != EventPhase.AT_TARGET) {
        // ...then a descendant is about to be removed from container
        trace("a descendant was removed from container: ", e.target);
    } else {
        // ...otherwise, container is about to be removed from its parent
        trace("container is about to be removed from its parent: ",
              + DisplayObject(e.target).parent);
    }
}
```

For reference, Example 20-6 presents the preceding `Event.ADDED` and `Event.REMOVED` example code within the context of a test class, `ContainmentEventDemo`. We'll study real-world containment-event examples over the next two sections.

**Example 20-6. Containment events demonstrated**

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

    public class ContainmentEventDemo extends Sprite {
        public function ContainmentEventDemo () {
            // Create Sprite objects
            var container:Sprite = new Sprite();
            var child:Sprite = new Sprite();
            var grandchild:Sprite = new Sprite();

            // Start listening for Event.ADDED and Event.REMOVED events
            container.addEventListener(Event.ADDED, addedListener);
            container.addEventListener(Event.REMOVED, removedListener);

            // Add child to container
            container.addChild(child); // Triggers addedListener() during
                                        // the bubbling phase

            // Add grandchild to child
            child.addChild(grandchild); // Triggers addedListener() during
                                        // the bubbling phase

            // Add container to Stage
            stage.addChild(container); // Triggers addedListener() during
                                        // the target phase

            // Remove grandchild from child
        }
    }
}
```
A Real-World Containment-Event Example

Now that we’ve seen how the Event.ADDED and Event.REMOVED events work in theory, let’s consider how they can be used in a real application. Suppose we’re writing a class, IconPanel, that manages the visual layout of graphical icons. The IconPanel class is used as one of the parts of a larger window component in a windowing interface. Any time a new icon is added to, or removed from, an IconPanel object, that object executes an icon-layout algorithm. To detect the addition and removal of child icons, the IconPanel object registers listeners for the Event.ADDED and Event.REMOVED events.

Example 20-7 shows the code for the IconPanel class, simplified to illustrate the use of Event.ADDED and Event.REMOVED. Notice that the Event.ADDED and Event.REMOVED
event listeners execute icon-layout code when the IconPanel gains or loses a new direct child only. No layout code is executed in the following situations:

- When an IconPanel object gains or loses a descendant that is not a direct child
- When an IconPanel object, itself, is added to a parent container

Example 20-7. Arranging icons in the IconPanel class

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

    public class IconPanel extends Sprite {
        public function IconPanel () {
            addEventListener(Event.ADDED, addedListener);
            addEventListener(Event.REMOVED, removedListener);
        }

        public function updateLayout ():void {
            // Execute layout algorithm (code not shown)
        }

        // Handles Event.ADDED events
        private function addedListener (e:Event):void {
            if (DisplayObject(e.target.parent) == e.currentTarget) {
                updateLayout();
            }
        }

        // Handles Event.REMOVED events
        private function removedListener (e:Event):void {
            if (DisplayObject(e.target.parent) == e.currentTarget) {
                updateLayout();
            }
        }
    }
}
```

The ADDED_TO_STAGE and REMOVED_FROM_STAGE Events

As discussed in the previous two sections, the Event.ADDED and Event.REMOVED events occur when a DisplayObject instance is added to, or removed from, a DisplayObjectContainer instance. The Event.ADDED and Event.REMOVED events do not, however, indicate whether a given object is currently on the display list. To detect when a DisplayObject instance is added to, or removed from, the display list, we use the Event.ADDED_TO_STAGE and Event.REMOVED_FROM_STAGE events, both of which were added to the display API with the release of Flash Player 9.0.28.0.

When a display object (or one of its ancestors) is added to the display list, the Flash runtime dispatches an Event.ADDED_TO_STAGE event targeted at that object. Conversely, when a display object (or one of its ancestors) is about to be removed from
the display list, the Flash runtime dispatches an Event.REMOVED_FROM_STAGE event targeted at that object.

Unlike the Event.ADDED and Event.REMOVED events, Event.ADDED_TO_STAGE and Event.REMOVED_FROM_STAGE events do not bubble. To receive an Event.ADDED_TO_STAGE or Event.REMOVED_FROM_STAGE event through an object’s ancestor, register with that ancestor for the event’s capture phase.

The generalized code required to register a listener with a DisplayObject instance for the Event.ADDED_TO_STAGE event is as follows:

```javascript
theDisplayObject.addEventListener(Event.ADDED_TO_STAGE, addedToStageListener);
```

The generalized event-listener code required for an Event.ADDED_TO_STAGE listener is:

```javascript
private function addedToStageListener (e:Event):void {
}
```

The generalized code required to register a listener with a DisplayObject instance for the Event.REMOVED_FROM_STAGE event is as follows:

```javascript
theDisplayObject.addEventListener(Event.REMOVED_FROM_STAGE, removedFromStageListener);
```

The generalized event-listener code required for an Event.REMOVED_FROM_STAGE listener is:

```javascript
private function removedFromStageListener (e:Event):void {
}
```

Display objects typically use the Event.ADDED_TO_STAGE event to ensure that the Stage object is accessible before using its methods, variables, or events. For example, suppose we’re creating a class, CustomMousePointer, that represents a custom mouse pointer. Our CustomMousePointer class extends the Sprite class so that its instances can be added to the display list. In the class, we want to register with the Stage instance for the MouseEvent.MOUSE_MOVE event so that we can keep the custom mouse pointer’s position synchronized with the system mouse pointer’s position. However, when a new CustomMousePointer object is created, it is initially not on the display list, so it has no access to the Stage instance and cannot register for the MouseEvent.MOUSE_MOVE event. Instead, the CustomMousePointer object must wait to be notified that it has been added to the display list (via the Event.ADDED_TO_STAGE event). Once the CustomMousePointer object is on the display list, its stage variable refers to the Stage instance, and it can safely register for the MouseEvent.MOUSE_MOVE event. The following code shows the relevant Event.ADDED_TO_STAGE excerpt from the CustomMousePointer class. For the full CustomMousePointer class code listing, see the section “Finding the Mouse Pointer’s Position” in Chapter 22.
package {
    public class CustomMousePointer extends Sprite {
        public function CustomMousePointer () {
            // Ask to be notified when this object is added to the display list
            addEventListener(Event.ADDED_TO_STAGE, addedToStageListener);
        }

        // Triggered when this object is added to the display list
        private function addedToStageListener (e:Event):void {
            // Now its safe to register with the Stage instance for
            // MouseEvent.MOUSE_MOVE events
            stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
        }
    }
}

Custom Event.ADDED_TO_STAGE and Event.REMOVED_FROM_STAGE events

The initial release of Flash Player 9 did not offer either the Event.ADDED_TO_STAGE event or the Event.REMOVED_FROM_STAGE events. However, using the original display API and a little ingenuity, we can manually detect when a given object has been added to or removed from the display list. To do so, we must monitor the state of that object’s ancestors using the Event.ADDED and Event.REMOVED events.

Example 20-8, which follows shortly, shows the approach. In the example, the custom StageDetector class monitors a display object to see when it is added to, or removed from, the display list. When the object is added to the display list, StageDetector broadcasts the custom StageDetector.ADDED_TO_STAGE event. When the object is removed from the display list, StageDetector broadcasts the custom StageDetector.REMOVED_FROM_STAGE event.

The StageDetector class’s custom ADDED_TO_STAGE and REMOVED_FROM_STAGE events can be used without any knowledge or understanding of the code in the StageDetector class. However, the StageDetector class serves as an interesting summary of the display list programming techniques we’ve seen in this chapter, so let’s take a closer look at how it works.

In the StageDetector class, the object being monitored for ADDED_TO_STAGE and REMOVED_FROM_STAGE events is assigned to the watchedObject variable. The root of watchedObject’s display hierarchy is assigned to the watchedRoot variable. The general approach taken by StageDetector to detect whether watchedObject is on the display list is as follows:

• Monitor the watchedRoot for Event.ADDED and Event.REMOVED events.
• Any time watchedRoot is added to a DisplayObjectContainer object, check if watchedObject is now on the display list (watchedObject is on the display list if its stage variable is non-null.) If watchedObject is now on the display list, dispatch the StageDetector.ADDED_TO_STAGE event. If it’s not, start monitoring the new watchedRoot for Event.ADDED and Event.REMOVED events.
• While watchedObject is on the display list, if the watchedRoot or any of the
watchedRoot’s descendants are removed from a DisplayObjectContainer object,
then check if the removed object is an ancestor of watchedObject. If the removed
object is a watchedObject ancestor, dispatch the StageDetector.REMOVED_FROM_
STAGE event, and start monitoring the watchedObject’s new display hierarchy root
for Event.ADDED and Event.REMOVED events.

The code for the StageDetector class follows.

Example 20-8. Custom ADDED_TO_STAGE and REMOVED_FROM_STAGE events

package {
    import flash.display.*;
    import flash.events.*;

    // Monitors a specified display object to see when it is added to or
    // removed from the Stage, and broadcasts the corresponding custom events
    // StageDetector.ADDED_TO_STAGE and StageDetector.REMOVED_FROM_STAGE.

    // USAGE:
    // var stageDetector:StageDetector = new StageDetector(someDisplayObject);
    // stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
    //                                addedToStageListenerFunction);
    // stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
    //                                removedFromStageListenerFunction);

    public class StageDetector extends EventDispatcher {
        // Event constants
        public static const ADDED_TO_STAGE:String = "ADDED_TO_STAGE";
        public static const REMOVED_FROM_STAGE:String = "REMOVED_FROM_STAGE";

        // The object for which ADDED_TO_STAGE and REMOVED_FROM_STAGE events
        // will be generated
        private var watchedObject:DisplayObject = null;

        // The root of the display hierarchy that contains watchedObject
        private var watchedRoot:DisplayObject = null;

        // Flag indicating whether watchedObject is currently on the
        // display list
        private var onStage:Boolean = false;

        // Constructor
        public function StageDetector (objectToWatch:DisplayObject) {
            // Begin monitoring the specified object
            setWatchedObject(objectToWatch);
        }

        // Begins monitoring the specified object to see when it is added to or
        // removed from the display list
        public function setWatchedObject (objectToWatch:DisplayObject):void {
            // Track the object being monitored
            watchedObject = objectToWatch;

            // Begin monitoring the object
            setWatchedObject(objectToWatch);

            // Dispatch the ADDED_TO_STAGE event
            dispatchEvent(new Event(StageDetector.ADDED_TO_STAGE));

            // Begin listening for removed events
            onRemovedFromStage = onRemovedFromStage;

            // Dispatch the REMOVED_FROM_STAGE event
            dispatchEvent(new Event(StageDetector.REMOVED_FROM_STAGE));
        }
    }

    // Example usage
    var stageDetector:StageDetector = new StageDetector(someDisplayObject);
    stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
        addedToStageListenerFunction);
    stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
        removedFromStageListenerFunction);
}
Example 20-8. Custom ADDED_TO_STAGE and REMOVED_FROM_STAGE events (continued)

// Note whether watchedObject is currently on the display list
if (watchedObject.stage != null) {
    onStage = true;
}

// Find the root of the display hierarchy containing the
// watchedObject, and register with it for ADDED/REMOVED events.
// By observing where watchedObject's root is added and removed,
// we'll determine whether watchedObject is on or off the
// display list.
setWatchedRoot(findWatchedObjectRoot());

// Returns a reference to the object being monitored
public function getWatchedObject ():DisplayObject {
    return watchedObject;
}

// Frees this StageDetector object's resources. Call this method before
// discarding a StageDetector object.
public function dispose ():void {
    clearWatchedRoot();
    watchedObject = null;
}

// Handles Event.ADDED events targeted at the root of
// watchedObject's display hierarchy
private function addedListener (e:Event):void {
    // If the current watchedRoot was added...
    if (e.eventPhase == EventPhase.AT_TARGET) {
        // ...check if watchedObject is now on the display list
        if (watchedObject.stage != null) {
            // Note that watchedObject is now on the display list
            onStage = true;
            // Notify listeners that watchedObject is now
            // on the display list
            dispatchEvent(new Event(StageDetector.ADDED_TO_STAGE));
        }
    }
}

// watchedRoot was added to another container, so there's
// now a new root of the display hierarchy containing
// watchedObject. Find that new root, and register with it
// for ADDED and REMOVED events.
setWatchedRoot(findWatchedObjectRoot());

// Handles Event.REMOVED events for the root of
// watchedObject's display hierarchy
private function removedListener (e:Event):void {
    // If watchedObject is on the display list...
    if (onStage) {
        // ...check if watchedObject or one of its ancestors was removed
Example 20-8. Custom ADDED_TO_STAGE and REMOVED_FROM_STAGE events (continued)

```javascript
var wasRemoved:Boolean = false;
var ancestor:DisplayObject = watchedObject;
var target:DisplayObject = DisplayObject(e.target);
while (ancestor != null) {
    if (target == ancestor) {
        wasRemoved = true;
        break;
    }
    ancestor = ancestor.parent;
}

// If watchedObject or one of its ancestors was removed...
if (wasRemoved) {
    // ...register for ADDED and REMOVED events from the removed
    // object (which is the new root of watchedObject's display
    // hierarchy).
    setWatchedRoot(target);

    // Note that watchedObject is not on the display list anymore
    onStage = false;

    // Notify listeners that watchedObject was removed from the Stage
    dispatchEvent(new Event(StageDetector.REMOVED_FROM_STAGE));
}
}

// Returns the root of the display hierarchy that currently contains
// watchedObject
private function findWatchedObjectRoot ():DisplayObject {
    var watchedObjectRoot:DisplayObject = watchedObject;
    while (watchedObjectRoot.parent != null) {
        watchedObjectRoot = watchedObjectRoot.parent;
    }
    return watchedObjectRoot;
}

// Begins listening for ADDED and REMOVED events targeted at the root of
// watchedObject's display hierarchy
private function setWatchedRoot (newWatchedRoot:DisplayObject):void {
    clearWatchedRoot();
    watchedRoot = newWatchedRoot;
    registerListeners(watchedRoot);
}

// Removes event listeners from watchedRoot, and removes
// this StageDetector object's reference to watchedRoot
private function clearWatchedRoot ():void {
    if (watchedRoot != null) {
        unregisterListeners(watchedRoot);
        watchedRoot = null;
    }
}
```
In Chapter 22, we’ll see the custom `StageDetector.ADDED_TO_STAGE` and `StageDetector.REMOVED_FROM_STAGE` events used in the `CustomMousePointer` class.

We’ve now finished our look at the container API. Now let’s consider one last short, but fundamental display programming topic: custom graphical classes.

### Custom Graphical Classes

We’ve drawn lots of rectangles, circles, and triangles in this chapter. So many, that some of the examples we’ve studied have had a distinct “code smell”: their code was repetitive, and therefore error-prone.

Learn more about code smell (common signs of potential problems in code) at [http://xp.c2.com/CodeSmell.html](http://xp.c2.com/CodeSmell.html).

To promote reuse and modularity when working with primitive shapes, we can move repetitive drawing routines into custom classes that extend the `Shape` class. Let’s start with a custom `Rectangle` class, using an extremely simple approach that provides a very limited set of stroke and fill options, and does not allow the rectangle to be changed once drawn. Example 20-9 shows the code. (We’ll expand on the `Rectangle` class’s features in Chapter 25.)

#### Example 20-9. Rectangle, a simple shape subclass

```java
package {
    import flash.display.Shape;

    public class Rectangle extends Shape {
        public function Rectangle (w:Number, h:Number, lineThickness:Number, lineColor:uint, fillColor:uint) {
```
Because Rectangle extends Shape, it inherits the Shape class's graphics variable, and can use it to draw the rectangular shape.

To create a new Rectangle, we use the following familiar code:

```javascript
var rect:Rectangle = new Rectangle(100, 50, 3, 0xFF0000, 0x0000FF);
```

Because Shape is a DisplayObject descendant, Rectangle inherits the ability to be added to the display list (as does any descendant of DisplayObject), like this:

```javascript
someContainer.addChild(rect);
```

As a descendant of DisplayObject, the Rectangle object can also be positioned, rotated, and otherwise manipulate like any other displayable object. For example, here we set the Rectangle object’s horizontal position to 15 and vertical position to 30:

```javascript
rect.x = 15;
rect.y = 30;
```

And the fun doesn’t stop at rectangles. Every class in the display API can be extended. For example, an application could extend the TextField class when displaying a customized form of text. Example 20-10 demonstrates, showing a TextField subclass that creates a hyperlinked text header.

Example 20-10. ClickableHeading, a TextField subclass

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

    public class ClickableHeading extends TextField {
        public function ClickableHeading (headText:String, URL:String) {
            html = true;
            autoSize = TextFieldAutoSize.LEFT;
            htmlText = "<a href='" + URL + ">
            " + headText + ">
            " + htmlText + ";
            border = true;
            background = true;
        }
    }
}
```

Here’s how we might use the ClickableHeading class in an application:

```javascript
var head:ClickableHeading = new ClickableHeading("Essential ActionScript 3.0",
                                             "http://www.moock.org/eas3");

addChild(head);
Figure 20-17 shows the resulting on-screen content. When the example runs in a Flash runtime, the text is linked to the companion web site for this book.

We’ll see lots more examples of display subclasses in the upcoming chapters. As you conceive of the visual assets required by your applications, consider the possibility of extending an existing display class rather than writing classes from scratch.

**Go with the Event Flow**

By now, you should feel relatively comfortable creating displayable content and adding it to the screen. Many of the examples in this book rely heavily on the fundamentals that were covered in this chapter, so you’ll have plenty of opportunities to review and expand on what you’ve learned. In the next chapter, we’ll learn how ActionScript 3.0’s event architecture caters to objects on the display list.
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**
oreillynet.com 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.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.