Roger Braunstein

ActionScript® 3.0

Second Edition

Develop for web and mobile devices

Enhance desktop apps with HD Video and 3D

Write AS3 for use with AIR®, Flex®, and Flash®

Bible

The book you need to succeed!

ActionScript® 3.0 Bible

ActionScript® 3.0 Bible

Second Edition

Roger Braunstein

About the Author

Roger Braunstein is a multidisciplinary programmer and the Chief Roger Officer of partlyhuman inc., an independent development and design shop in Brooklyn, NY. Roger is an author of the first edition of *ActionScript 3.0 Bible,* and a short e-book, *Introduction to Flex 2.* Roger builds games, mobile apps, interactive art, tools, web apps, and websites using any technology he can wrap his head around. In just the past two years he's built stuff for LEGO, MoMA, Tim Burton, Antony and the Johnsons, AOL, HBO, Puma, General Mills, Sport Chalet, Pepsi, Coke, OMGPOP, the French Culinary Institute, and more. Additionally, he keeps busy with a series of independent projects. When not in front of a computer, Roger enjoys normal human activities such as biking, cooking, reading, traveling, taking photos, roller-skating, and dancing to music made on GameBoys. He is perpetually too preoccupied to put anything interesting on his site `http://partlyhuman.com/`, but you can use it to get in touch with him.

Credits

Acknowledgments

This book, like all gargantuan projects, was a team effort. Its pages were touched by many hands, at Wiley and elsewhere. Losing even a single pair of these hands would mean you wouldn't have this book in yours right now, so I want to thank each and every person on my team, whether I sent you 20 e-mails a day or never met you.

My most heartfelt thanks go to my project editor Brian MacDonald, and my acquisitions editor Scott Meyers, who were nothing short of bodacious to work with. When I think of how many times I pushed for some new, unconventional feature or change to the established Bible series, and then look at this book and see all of my ideas in print, I know it's Brian and Scott who have somehow done their magic, convinced the right people, and made it happen. Thank you.

Perhaps every author forms a good relationship with his project editor just because he's the person who's there from the first day of writing through the last, exhausted round of edits. But I think Brian MacDonald is exceptional. I've never worked with an editor half as good. He had the right solution for every problem. He was behind me and my ideas through thick and thin. His edits were always spot-on. He was always available, always funny, and he even got my nerdy jokes. Trust me, if you're writing a programming book, bribe whoever you need to hire Brian.

I can't talk about my team without mentioning the exhaustive copy editing done by Karen Gill. She kept tabs on all my writing, cut down my useless tiresome redundant logorrhea, and made this book less of a slog to get through. Trust me, you owe her one. I have the feeling her job is a little thankless, but I do thank her.

Caleb Johnston, besides being a good friend, was a terrific technical editor. Somehow, with a full course load for his master's degree, he managed to find the time to review this entire book by himself. (The previous edition, although smaller, had three technical editors!) More than just scrutinizing it, he made detailed and thoughtful criticisms, with as much ruthlessness as I demanded of him. He found some quirky errors, and because he found them, you won't have to.

Special thanks go out to my friend Corey Lucier at Adobe, who was kind enough to answer a few questions about the internal workings of Flash Player or forward them on to Flash Player engineers. Their additions helped me be even more accurate in this book, and Corey's help was invaluable.

Special thanks also go to Whitney Gardner, who made two sublime illustrations for this book when I was at my wit's end.

Thanks to the U.S. economy for making my day job so unprofitable it made sense to sit in my room and write for nine months. Thanks to coffee: you complete me. Thanks to my crazy SafeType keyboard, which saved me from crippling arm pain. Thanks to my friends for, I dunno, whatever, I like you.

Last but not least, thanks and lotsa lotsa love to my parents.

Contents at a Glance

Contents at a Glance

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Part IV: Event-Driven Programming   407

Chapter 20: Events and the Event Flow 409

Chapter 21: Interactivity with the Mouse and Keyboard 429

Contents

Part V: Error Handling   489

Chapter 24: Errors and Exceptions 491

Chapter 25: Using the AVM2 Debugger 505

Chapter 26: Making Your Application Fault-Tolerant 523

Contents

Contents

Contents

Contents

xxxiii

Contents

Contents

DIAGRAM OF CLASSES
The Flash Player 10.1 API by Chapter Number

Top Level

| Object | 10 |

Boolean	2		Array	8		Number	7
Function	3		Vector	9		int	7
Class	4		XML	11		uint	7
String	6		XMLList	11		Date	7
RegExp	12		Error	24		Math	7

flash.display

| DisplayObject | |

| Shape | 35 | | Bitmap | 36 | | AVM1Movie | 27 |
| | | | BitmapData | 36 | | | |

| InteractiveObject | |

| | | | | | | SimpleButton | 14 |

| DisplayObjectContainer | |

Sprite	14		Stage	14		Loader	27
MovieClip	14					LoaderInfo	27
Graphics	35		Shader	38		ShaderInput	38
			ShaderJob	38		ShaderParameter	38

flash.events

| EventDispatcher | 20 |

| Event | 20 |

MouseEvent	21		TouchEvent	23		TimerEvent	22
KeyboardEvent	21		GestureEvent	23		FullScreenEvent	14
FocusEvent	21		AccelerometerEvent	23		ProgressEvent	27
ContextMenuEvent	21		ActivityEvent	33		NetStatusEvent	32

| ErrorEvent | 20 |

| IOErrorEvent | 27 | | SecurityErrorEvent | 27 | | UncaughtErrorEvent | 24 |

flash.utils

| ByteArray | 13 | Dictionary | 10 | Timer | 22 |

flash.geom

Point	14	Vector3D	34	Transform	34
Matrix	34	Matrix3D	34	ColorTransform	34
Rectangle	14	PerspectiveProjection	15	Utils3D	40

flash.text — 17

- TextField
- StyleSheet
- TextFormat
- Font

flash.text.engine — 18

- TextLine
- TextBlock
- ContentElement
- FontDescription

flash.textLayout — 18

- TextFlow
- StandardFlowController
- ContainerController
- FlowElement

flash.net

URLRequest	27	Socket	28	FileReference	30
URLLoader	27	XMLSocket	28	FileReferenceList	30
URLVariables	27	NetConnection	32	FileFilter	30
LocalConnection	44	NetStream	32	SharedObject	29

flash.media

| Sound | 31 | SoundChannel | 31 | SoundMixer | 31 |
| Video | 32 | Camera | 33 | Microphone | 33 |

flash.ui

| Keyboard | 21 | Mouse | 21 | Multitouch | 23 |

flash.printing — 19

- PrintJob

flash.external — 43

- ExternalInterface

flash.filters — 37

- BitmapFilter
- BlurFilter
- DropShadowFilter
- DisplacementMapFilter

flash.globalization — 41

- DateTimeFormatter
- CurrencyFormatter
- Collator
- StringTools

Introduction

Hello! Welcome to the Second Edition of *ActionScript 3.0 Bible*. This book aims to be a comprehensive resource on all things ActionScript 3.0. You can use it as an instructional textbook, as a reference book, or both. The Bible introduces concepts used in ActionScript 3.0 with both background information and real-world examples to enrich your understanding. Armed with the knowledge contained herein, you will be able to bring your ideas to life with ActionScript 3.0, whether they are applications, mobile apps, web sites, presentations, games, toys, art, or experiments.

ActionScript 3.0 is the language used in a range of tools developed by Adobe, including Flash Professional, the Flex framework, Flash Builder (formerly Flex Builder), AIR, and Flash Lite 4. This book's coverage remains agnostic of the tool you choose to use. In general, it will not cover topics specific to a single tool, but it will cover the commonality they all share: ActionScript 3.0. When something is handled differently between different tools, I let you know how. Accordingly, this is the right book for you to learn how to program with any of these tools.

What's New in This Edition

This book, the second edition of *ActionScript 3.0 Bible*, is the result of countless hours of revising, writing, drawing, editing, and yes, programming. I've exhaustively and repeatedly gone over the first edition, as well as feedback from readers, to make what I hope is the most comprehensive, easy-to-use, and up-to-date ActionScript 3.0 programming book out there.

Most importantly, this book covers the latest improvements and additions to ActionScript 3.0 up through Flash Player 10.1. I've added seven new chapters and updated every other chapter for these new features. The list of new technology and new possibilities that have opened up since the first edition was published is huge, but I'm most excited about Flash Player's native 3D API, Pixel Bender shaders, the new Text Layout Framework, multitouch capability, hardware accelerated graphics, and the `Vector` class.

Since the first edition, ActionScript 3.0 has exploded onto embedded devices (like televisions and set-top boxes) and mobile phones. Additions to ActionScript like hardware-accelerated graphics, multitouch, accelerometer support, and geolocation services make these applications possible and really exciting. Flash Player 10.1 content can or soon will be able to run on mobile phones running Palm WebOS, Android, Windows Mobile, and Blackberry OS. You can compile your apps for the iPhone using Flash Professional CS5. Other non-smartphone mobiles can run Flash Lite 4, which also uses ActionScript 3.0. As a developer myself, I'm practically frothing at the mouth to develop for these systems, and in this edition, I share the tools to do so.

I put a big emphasis on examples in this edition. I've torn apart almost all the old code from the previous edition and added around 250 new examples. And I've made it easier than ever to run and play with example code in this edition. Just type in the URL included with each example (like `http://actionscriptbible.com/ch40/ex6`), and you can see the code with a screenshot, run the example, or download the code, all with a single click! Better yet, you can "fork" the example to

play with the code, all with zero setup and zero software required! The section "Running Example Code" in this Preface shows you how. I think you'll be amazed at how easy and fun it is to use examples in this edition.

ActionScript 3.0 was introduced in 2006. Because it's not so hot-off-the-grill anymore, I've retooled this second edition slightly. It's no longer an upgrade guide from ActionScript 2.0, although I do provide a few notes for those readers in Chapter 1, "Introducing ActionScript 3.0."

Last but not least, this is a real second edition. I haven't just added to the *ActionScript 3.0 Bible*, I've rewritten it ruthlessly and repeatedly to make it easier to read, more informative, and accurate. I've made thousands of changes throughout the book, and I think this edition is really the best, most comprehensive, and easiest to read book out there on ActionScript 3.0 and the Flash Platform.

Make Sure You Buy the Right Book

ActionScript 3.0 is the third major incarnation of the ActionScript language. Previously, ActionScript was used primarily to develop Flash content. Now ActionScript is being applied in an increasing variety of contexts, and it is a mature technology in its own right. The *ActionScript 3.0 Bible* was written with this in mind, while additional offerings from Wiley provide focused training for the tools that use ActionScript 3.0.

The Flash Bible series continues with the *Flash CS4 Professional Bible* by Robert Reinhardt and Snow Dowd. Rely on the *Flash CS4 Professional Bible* to learn the ins and outs of the latest version of Flash such as using the timeline, the library, and drawing tools. The Flash Bible series can get you started with ActionScript, but it is not a comprehensive reference for programming topics in Flash. If you plan on using Flash Professional to develop your ActionScript projects, I recommend using the *Flash CS4 Professional Bible* as a complement to this book.

If you are using ActionScript 3.0 with Flex, I recommend complementing this book with the *Flex 3 Bible* by David Gassner or, for a more approachable introduction, *Adobe Flex 3.0 For Dummies* by Doug McCune and Deepa Subramaniam. Flex is based on ActionScript 3.0, so this book will get you well on your way to being a skilled Flex programmer. The best Flex programmers can dive into the Flex framework code, which is all AS3.

If you are developing desktop applications using Adobe AIR, this book will give you an excellent foundation. Everything in this book applies to AIR as well, but you'll need a definitive guide like the *Adobe AIR Bible* (by Benjamin Gorton, Ryan Taylor, and Jeff Yamada) to cover AIR-specific features I leave out of this book.

How This Book Is Organized

The name *ActionScript 3.0*, as commonly used, refers to both a programming language and the Flash Player API, or Application Programming Interface. Mastering ActionScript 3.0 the *language* will let you write programs, but it's the Flash Player API — an enormous library of classes and functions — that will let you create programs that you can see, that move, that react to the keyboard and mouse, that play video and sound, that connect to the internet, and more. All the subject matter of the ActionScript 3.0 language, the Flash Player API, and related topics such as debugging and embedding are organized logically into parts of this book. The parts are further divided into chapters, each of which focuses on a single topic. Of course, the Table of Contents and Diagram of Classes, as well as the index, will help you find what you're looking for.

If you're new to ActionScript 3.0, this book works well from front to back. However, after learning the language in Part I, "ActionScript 3.0 Language Basics," and for readers with some AS3 experience, you can jump around as you desire to the topic that you'd like to learn about. Topics may require knowledge of preceding chapters, but these are cross-referenced for your benefit.

Part I: ActionScript 3.0 Language Basics

Part I of this book is designed to get all readers up to the same level. This part introduces all the different parts of the Flash Platform and how they connect, and it demystifies the differences between Flash Professional and Flash Player, ActionScript 3.0 and the API, and the compiler and the debugger. It teaches you what ActionScript 3.0 is all about and how to use the language. By the end of this part you should be able to write programs in ActionScript 3.0.

If ActionScript 3.0 is the first programming language you have used, make sure you read all of Part I. Everyone but experienced Flash Platform developers should read Chapter 1, "Introducing ActionScript 3.0." If you have experience with object-oriented programming languages other than ActionScript, you should at least skim Part I to get a feel for the syntax and features of the language.

After Part I and a bit of practice, readers new to ActionScript and readers with prior experience should be equally prepared to tackle the rest of the book.

Part II: Core ActionScript 3.0 Data Types

This part covers the data structures in ActionScript 3.0 that you will use most commonly, how they are implemented in ActionScript, and what you can do with them. It's an essential read. Experienced programmers can benefit from Chapter 9, "Vectors," on this new data type for Flash Player 10.

Part III: The Display List

This part covers the ground you'll need to start making programs that have a visual aspect. The fundamentals introduced here are used frequently in other parts. Experienced programmers will enjoy the addition of Chapter 15, "Working in Three Dimensions," and Chapter 18, "Advanced Text Layout."

Part IV: Event-Driven Programming

This part provides the fundamentals for writing interactive software by introducing events. This is a key topic that will be used in every following part. This edition adds Chapter 23, "Multitouch and Accelerometer Input."

Part V: Error Handling

This part gives you the tools you need to write rock-solid programs in ActionScript 3.0, including error handling and debugging.

Part VI: External Data

In this part, you'll learn about using information from the internet; communicating with servers; and sending, receiving, and saving files.

Part VII: Sound and Video

This part explains how to add multimedia capabilities to your program. You'll learn how to load, stream, play, capture, and control both audio and video. You'll also get to synthesize sound with code.

Part VIII: Graphics Programming and Animation

In this part, you'll learn how to use ActionScript 3.0 to generate visuals and make them move. This part is expansive and genuinely fun. You'll build all kinds of effects, games, and even a 3D engine. Experienced programmers will sink their teeth into Chapter 38, "Writing Shaders with Pixel Bender," and Chapter 40, "Advanced 3D."

Part IX: Flash in Context

This final part discusses real-world concerns that intersect with Flash Player, such as embedding and deploying your code, interfacing with code in other instances of Flash Player and in the host browser, and making your application usable by everyone. Experienced programmers should read Chapter 41, "Globalization, Accessibility, and Color Correction."

Conventions and Features

There are many different organizational and typographical features throughout this book designed to help you get the most from the information.

Icons

Throughout this book you will see sections called out from the main text. I use these special notes to bring important information to your attention or to add context and optional additions to the main discussion.

Caution

Cautions let you know about common trip-ups and things to think about — not always related to ActionScript — while writing your code. ∎

Tip

Tips are used to provide information that can make your life easier. ∎

Note

Notes provide ancillary information that is useful but that you won't suffer from skipping. They sometimes refer you to further reading. ∎

Version

These callouts bring to your attention any code or features that may depend on the version of the runtime or the compiler. If you use them in incompatible versions, they either won't compile or won't run properly.

Many new features are covered in this edition that have been added since Flash Player 9. The version callout lets you know where they can be used. These version callouts are only used when the requirement is greater than Flash Player 9.0. ■

Product Conventions

There are a bunch of products out there that work with ActionScript 3.0, and a bunch of versions thereof. I refer to these by consistent and generic names:

- Flash Builder — I've written this edition using Adobe Flash Builder 4 Premium Beta. When I use the term "Flash Builder," it applies generically to Adobe Flash Builder 4, as well as Adobe Flex Builder 3.

- Flash Professional — While writing I used prerelease versions of Adobe Flash Professional CS5. The term "Flash Professional" applies to Adobe Flash Professional CS5, Adobe Flash CS4 Professional, and Adobe Flash CS3 Professional. Many of the latest features covered in this book require CS4 or CS5. You can only use a given feature if your copy of Flash Professional can publish SWFs for the Flash Player version noted in the text.

When there are even more recent versions of these tools out, what I've said about them will probably apply to the new version, but obviously there are no guarantees.

Official Documentation

Although this edition of *ActionScript 3.0 Bible* is terrifically deep and detailed, the official Adobe documentation is still an indispensible resource for any ActionScript programmer. It's organized for faster access to methods and classes, so it's not a bad idea to keep it open while you program along with this book.

In some cases, I refer to the documentation, calling it the *AS3LR*. This is for the "ActionScript 3.0 Language Reference," although your copy may be called the "ActionScript 3.0 Language and Components Reference" or the "ActionScript 3.0 Reference for the Adobe Flash Platform." You may have a local copy of this reference that was included with Flash Professional or Flash Builder. You can also access the latest version online, at `http://bit.ly/as3lr`.

In its latest incarnation, you can select the products and runtimes that are shown in the documentation. This book covers the Flash Player 10.1 runtime and none of the extra products. Configure the filters to match these settings and you'll see all the classes covered in this book.

Code Formatting and Conventions

ActionScript code is used heavily throughout this book. Text that appears in code or is for use in code is written in a monospace font, like `Object`. In addition, example code is included in two ways: examples and snippets.

Examples, while concise, are self-contained, complete programs. They are provided to illustrate a complete task achieved with code. Although the task is usually very small, it should illustrate the utility of the topic and demonstrate what you're learning in the text surrounding the example. Because examples include all required setup and infrastructure, you can run them without intervention. The section "Running Example Code" shows you how.

Examples always have at least one full class. The class exists in the default package and corresponds to the number of the example. The first example in the sixth chapter is called Example 6-1, and its class

name is `ch6ex1`. You can always find it online at `http://actionscriptbible.com/ch6/ex1`. Examples look like this:

EXAMPLE 6-1 `http://actionscriptbible.com/ch6/ex1`

Escaping Strings

```
package {
  import com.actionscriptbible.Example;
  public class ch6ex1 extends Example {
    public function ch6ex1() {
      var porky:String = 'Porky says "That\'s all folks!"';
      trace(porky);
    }
  }
}
```

Snippets are short blocks of code — even as small as a single line — that are prevalent throughout the book. Snippets don't have the necessary structural code to be launched and compiled by themselves. They are to the point, including only the important content and leaving out the context. They can also be conversational, where one snippet follows a preceding one, either revising it or adding to it. The text around snippets lets you know what they are meant to accomplish.

Some snippets can be taken literally, whereas others are merely illustrative. Where multiple snippets continue through a chapter, they may be collected into one example heading and found online at one URL.

As you read the book, I recommend that you read the snippets but interact with the larger examples: modify them to do different things, play with them to see if they break, add onto them to make them yours. Getting hands-on with code is the best way to really learn it.

Running Example Code

This section will explain, step by step, how to get examples from this book running so you can experiment with them. You can run example code online or on your computer using the IDE of your choice. In this section I'll demonstrate running the code online, with Flash Builder 4, and with Flash Professional CS5. Of all these, running the code online is the easiest.

It's so important that you know how to run and use example code that I've distilled this information into screencasts for those of you who prefer to learn visually. Find these screencasts at `http://actionscriptbible.com/runningcode/`.

First, let's take a look at how to run a complete example. You can recognize examples because they are full classes with a name like `ch0ex1` that extend either `flash.display.Sprite` or `com.actionscriptbible.Example`.

The `Example` class is used only when necessary, mostly for the benefit of the online examples. It simply redirects `trace()` output to the screen so that you can see it no matter your setup.

Whenever you see this class, you can safely replace it with `Sprite`. Alternatively, visit the web site at `http://actionscriptbible.com/` to get the source for `Example`.

Let's look at how to run Example 0-1.

EXAMPLE 0-1   `http://actionscriptbible.com/ch0/ex1`

An Example Example

```
package {
  import com.actionscriptbible.Example;
  public class ch0ex1 extends Example {
    public function ch0ex1() {
      trace("Congratulations! You just ran Example 0-1!");
    }
  }
}
```

Online Examples

The online example system is Kayac's awesome and free *wonderfl* service. I want to give a special thanks to Masakazu Ootsuka for building and maintaining wonderfl and for working with me to provide online examples.

See a screencast version of this tutorial at `http://actionscriptbible.com/runningcode/online/`.

System Requirements

To run examples online, all you need is a reasonably modern browser and the Flash Player plug-in. I suggest you use the debugger version of the latest version of Flash Player you can get your hands on. You can get the latest public release at `http://get.adobe.com/flashplayer/`, but I strongly recommend you get a debugger version of the player at `http://www.adobe.com/support/flashplayer/downloads.html`. To run everything in the book, you're going to need Flash Player 10.1 or later.

Running Code

Running code online couldn't be simpler.

1. Use your browser to navigate to the link provided. Here it is: `http://actionscriptbible.com/ch0/ex1`.

2. On the left, you'll see the code of the example. On the right, you'll see a screenshot. To run the code, click the "Play" button that appears over the screenshot.

Your screen will look like Figure I-1.

Downloading Code
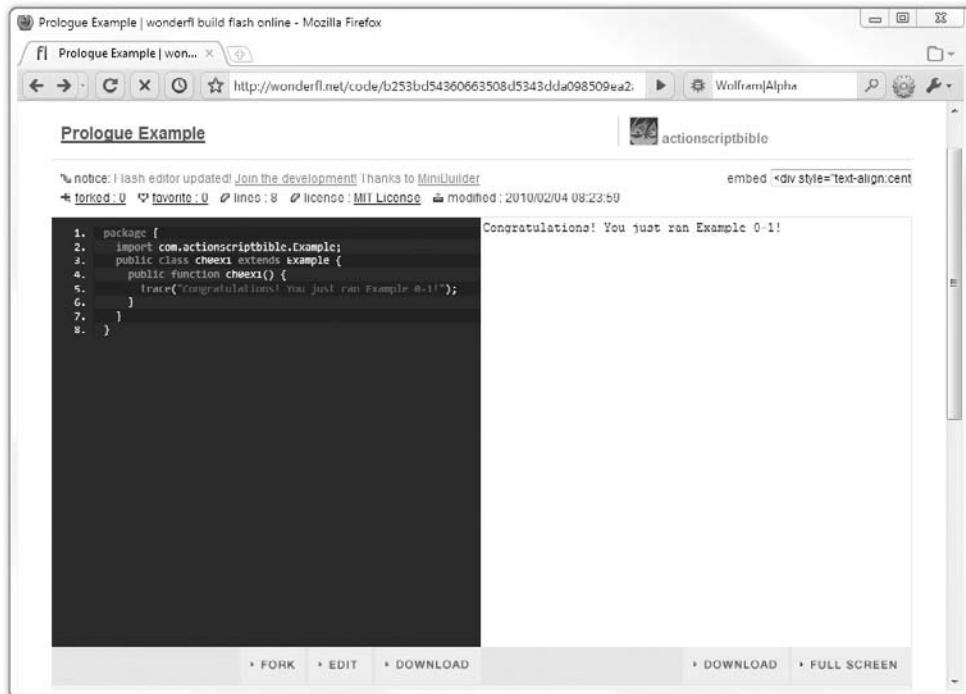
Just click the Download button below the code on the example page to get a copy of the source code to play with locally. Additionally, all example code is available on `http://actionscriptbible.com/downloads/`.

FIGURE I-1

Running an example online

Community Features

Wonderfl lets you participate with other readers across the world. You can build on the examples I've provided, ask questions, and favorite the code. To use these features, you'll first need to sign in at `http://wonderfl.net/login`. It's free and uses OpenID, so you don't need to remember yet another password. Nice!

To edit an example, click the Fork button below its source code. Then use the editor that appears in your browser to interactively write ActionScript code. Your creation will appear below my original example on the example page so that everyone else can benefit from your work! Make sure to title your work something descriptive and comment so that people know what you did.

Likewise, below each example code, you may see many forked versions that other readers created. Try looking at these to enhance your understanding of the topic. Of course, you can even fork code that's been forked from my original example. Keep the fire burning!

Under the Talk box on the example page, you can discuss the example. Keep it civil, folks. And I'm pretty busy, so don't expect me to personally reply to feedback here. Sorry!

Flash Builder

You can build examples from this book using Flash Builder (or Flex Builder; see "Product Conventions" earlier in this Preface). At the time of writing, Flash Builder 4 was not public yet, but I'll wager you can get a free 30-day trial version at `http://adobe.com/products/flex/`.

See a screencast version of this tutorial at `http://actionscriptbible.com/runningcode /flashbuilder/`.

System Requirements

Flash Builder 4 will be available for Linux, Windows, and Mac OS X. You can find the full system requirements at `http://adobe.com/products/flex/systemreqs/`.

Running Code

Running and editing code in Flash Builder can be a little tricky if you've never done it before, but in time it'll get to be second nature.

1. Launch Flash Builder.

2. Create a new ActionScript project using File ➪ New ➪ ActionScript Project. Give the project a name, and to make sure you can run all the code in the book, use the latest Flex SDK version possible (Flex 4.0 for code in this book). Place it in a convenient location. I'll name the project `as3b` and place it in `C:\Users\Roger\Documents\as3b\`.

3. Add the example file to the source directory. By default this is `src\` inside the project folder.

    a. To copy the file from the book, use File ➪ New ➪ ActionScript Class. In the dialog box, give it the name `ch0ex1`, and make it extend `flash.display.Sprite`. Keep the package blank. In the editor that appears, type in the example code as listed in the book.

    b. To download the code, get the example from `http://actionscriptbible.com /downloads/`, and place the file `ch0ex1.as` into the source directory. For me, that's `C:\Users\Roger\Documents\as3b\src\`. It should appear in the Package Explorer panel.

    c. Alternatively, you can download the code from the example page at `http://actionscriptbible.com/ch0/ex1`. Click the Download button on wonderfl, and move the file into the source directory. You'll also have to rename it to `ch0ex1.as`.

4. Once you have the file in your project, locate it in the Package Explorer. It should appear under your project, in the source directory. For me, it's `as3b\src\(default package) \ch01ex1.as`. Right-click the file, and choose Set as Default Application.

5. Click the Run or Debug buttons in the toolbar, or choose Run ➪ Run as3b or Run ➪ Debug as3b from the menu.
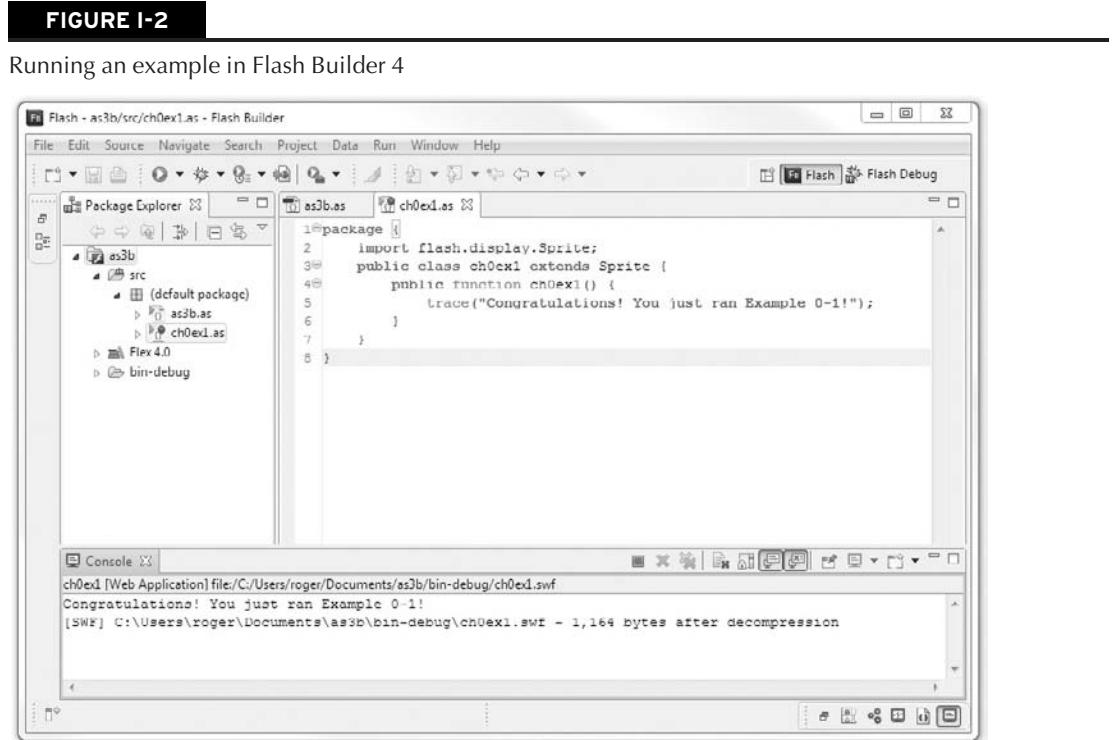
Note that you only need to follow Step 2 once. Once you have a project set up for your examples, you can keep adding examples to them. To get them to run, you will need to add the files to the source directory and add them as applications as in Steps 3–5. You don't need to remove older examples.

You can choose to change the `com.actionscriptbible.Example` class to `flash.display .Sprite` before running. Then trace output will appear in the Console panel, but not in Flash Player. Otherwise, you'll need the `Example` class in your source directory under `com\actionscriptbible`. For me, the full path would be `C:\Users\Roger\Documents \as3b\src\com\actionscriptbible\Example.as`.

Your screen should look like Figure I-2.

FIGURE I-2

Running an example in Flash Builder 4

By default, Flash Builder will launch the example in your web browser. To debug this, you must have the debugging version of Flash Player installed in the browser it uses. However, you also have the option of running the application in Flash Player without the browser, which I prefer. (For Part IX, "Flash in Context," you'll need to use the browser.) Open the project's properties by right-clicking it or choosing Project ➪ Properties in the menu. Choose the ActionScript Compiler section, uncheck the Generate HTML wrapper file, and click OK. After performing these steps, Flash Builder should run your application in the standalone Flash Player.

Flash Professional

You can also use Flash Professional to run example code from the book. To run every example, you'll need Flash Professional CS5 or later. At the time of writing, this was not available to the public, but I'll wager you can get the standard 30-day trial for free at http://www.adobe.com/products/flash/.

See a screencast version of this tutorial at `http://actionscriptbible.com/runningcode/flash/`.

System Requirements

Flash Professional CS5 will be available for Windows and Mac OS X. You can find the full system requirements at `http://www.adobe.com/products/flash/systemreqs/`.

Running Code

Running example code in Flash Professional isn't too hard. You'll need to follow a few steps.

1. Launch Flash Professional.

2. Create a new Flash file for ActionScript 3.0 by choosing File ➪ New ➪ Action-Script 3.0. Save this file into a directory somewhere. I'll create a new directory called `C:\Users\Roger\Documents\as3b\`. I'll call the file `example.fla`. These names are completely arbitrary and up to you.

3. Ensure that the Flash file is set to publish for the latest Flash Player version. If you did Step 1, this should already be set correctly, but to make sure, use File ➪ Publish Settings, select the Flash tab, and ensure that Player is set to Flash Player 10. Click OK.

4. Add the example file, `ch0ex1.as`, to the same directory as `example.fla`.

   a. To copy the file from the book, use File ➪ New ➪ ActionScript 3.0 Class. In the dialog box, give it the name `ch0ex1`. In the editor that appears, type in the example code as listed in the book. Save the file as `ch0ex1.as` in the same directory as `example.fla`.

   b. To download the code, get the example from `http://actionscriptbible.com/downloads/`, and place the file `ch0ex1.as` into the same directory as `example.fla`.

   c. Alternatively, you can download the code from the example page at `http://actionscriptbible.com/ch0/ex1`. Press the Download button on wonderfl, and move the file into the same directory as `example.fla`. You'll also have to rename it to `ch0ex1.as`.

5. Set the Document Class of `example.fla` to `ch0ex1`. Find the Class entry box in the Properties panel under Publish, and type `ch0ex1` into it. Don't include the `.as` part.

6. Test the file with Control ➪ Test Movie. You should see the output in the Output panel.
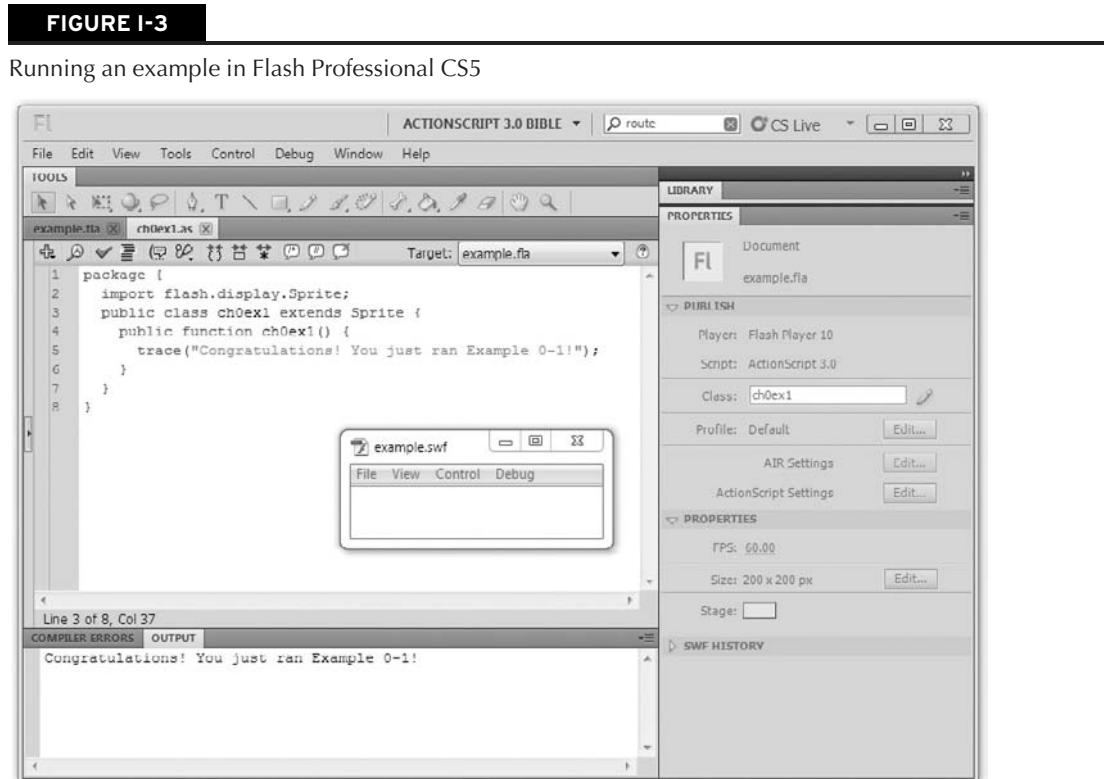
Note that you only need to follow Steps 2 and 3 once. Once you have a Flash file set up for your examples, you can keep adding example files to the directory and changing the Document Class of `example.fla`. For each new example, follow Steps 4–6.

You can choose to change the `com.actionscriptbible.Example` class to `flash.display.Sprite` before running. Then trace output will appear in the Output panel, but not in Flash Player. Otherwise, you'll need the `Example` class in your directory under `com\actionscriptbible`. For me, the full path would be `C:\Users\Roger\Documents\as3b\com\actionscriptbible\Example.as`.

Your screen should look like Figure I-3.

Running an example in Flash Professional CS5

Other Tools

You can use plenty of other tools to run example code, as explained in Chapter 1, "Introducing ActionScript 3.0," including the free, open-source version of the Flex SDK. I won't cover them here, but most other tools use a similar technique to Flash Builder's. If you're having trouble running example code, see the documentation for your product. This may also be a good time to reiterate that the easiest way to run example code is on the online version.

On the Web Site

This book has a companion web site at `http://actionscriptbible.com/`. The site provides resources for this book as well as bonus material:

- Downloads for all example code
- An index of examples
- An ActionScript 3.0 cheat sheet

- Most common ActionScript 3.0 mistakes
- Links to popular programming forums to find an answer for any ActionScript 3.0 question
- Links to give your feedback about this book
- More information about the handsome and stylish author

Where to Go from Here

Armed with the knowledge herein, you should be equipped to program in ActionScript 3.0. Countless doors open when you can implement your ideas in code, and there's no way I could tell you what your newfound knowledge could help you do. That's up to you. I strongly encourage you to work on projects aligned with your interests. You learn best by doing, and you do more when you do what you enjoy. So dream up an application, an animation, a simulation, a game, or an installation, and make it happen. Find some freelance jobs that sound enticing and bid for them. Challenge yourself, and keep the *ActionScript 3.0 Bible* by your side for reference.

Once you are comfortable programming in ActionScript 3.0, give the book to a new AS3 disciple and move on with your own learning! The field of programming is wonderfully wide. You can dive deeper: learn about algorithms, data structures, object-oriented design, design patterns, architecture, or software development practices. You can supplement your ActionScript 3.0 skills by learning Flex or AIR. You can make things realistic: learn computer graphics techniques, physics, animation, or algorithmic art. You can learn more about programming by learning a different kind of language. You can focus on server communication. Hey, you can even write a big fat programming book. Whatever you do, get to the chopper!

Part I

ActionScript 3.0 Language Basics

Introducing ActionScript 3.0

I n this chapter you'll look at what ActionScript is, where you can use it, and what you can do with it. You'll learn where ActionScript fits in the grand scheme of the Flash Platform, and you'll take a complete tour of the tools and technologies involved therein.

What Is ActionScript 3.0?

You may well already know the answer to this question, because you had enough interest in it to buy this book! ActionScript 3.0 is the language used to program interactive Flash content. Where this content goes and how you can build it is the subject of the following section.

ActionScript 3.0 is a well-organized, mature language that shares much of its syntax and methodologies with other object oriented, strongly typed languages, so an experienced programmer can readily pick it up. Don't fear, though, for this book introduces ActionScript from the bottom up and starts gently.

If you've used Flash before but never ActionScript, you might know that you can build content for Flash Player without ActionScript — but without ActionScript, Flash is just an animation tool (though, admittedly, a good one). ActionScript is necessary when you want to create Flash content that is highly dynamic, responsive, reusable, and customizable. Here's just a short list of the many things you can accomplish using ActionScript:

- Loading images
- Playing audio and video
- Drawing programmatically
- Loading data such as XML files
- Responding to user events such as mouse clicks

To get the most out of the Flash Platform, you're gonna need ActionScript 3.0. And as you learn what it can do, you'll be amazed at its power. But let's look at how ActionScript fits into the Flash universe.

Exploring the Flash Platform
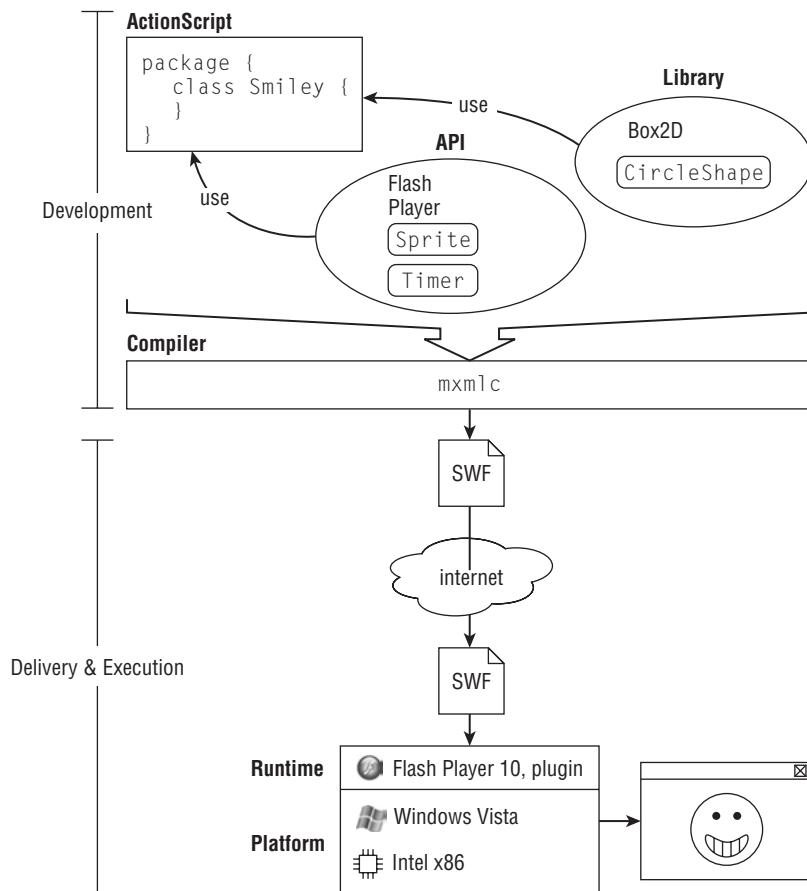
The wild world of Flash is by no means small, and wrapping your head around ActionScript and all the technologies related to it is challenging. Even if you don't know all the latest technologies in the peripheries, it's essential that you know major parts of the *Flash Platform* and how they work together. Defining that term and what it encompasses will be the goal of this section.

You'll examine the different parts of the Flash Platform from the perspectives of you building content and the user getting and running content. I'll define and discuss the tools, languages, platforms, and runtimes involved in the process. Figure 1-1 shows a bird's-eye view of the Flash Platform.

FIGURE 1-1

A high-level overview of the Flash Platform

You, the programmer, write in a computer programming *language*, ActionScript 3.0. You use *tools*, like Flash Builder, and specifically a *compiler*, to convert your code into an *executable*, in this case a *SWF*.

The SWF is delivered to the end user on her *platform* and executes inside a *runtime*, usually Flash Player, of which there are many versions. Let's break this down.

A Programmer's Perspective

For all the ways that the Flash Platform is unique, it shares the same basic steps as most programming environments. You write code in a language using some kind of editor, tool, or integrated development environment, and you use a compiler to convert that code into a file that can be run on the target environment.

Language

A computer programming *language* defines the grammar and lexicon that you'll be working in to create beautiful code. ActionScript 3.0, Python, Java, Lua, and C# are different languages. They all look different and have different rules for what you type where, what words are reserved, how to loop and how to write comments, and even where you can and can't put spaces. ActionScript 3.0 is not the same language as ActionScript 2.0, and neither is ActionScript 1.0.

ActionScript 3.0 has some features in common with modern JavaScript, because both are designed to adhere to specifications of a family of languages called ECMAScript. At the time of writing, this fact is little more than a curiosity, because most JavaScript in use is written to a baseline standard far behind the kind of JavaScript that starts to look like ActionScript 3.0. Furthermore, adherence to ECMAScript standards has provided little visible benefit, and progress marches on. In general, ActionScript 3.0 looks most like Java or C#. Coming to ActionScript 3.0 from either of these languages, or ActionScript 2.0, should be a fairly smooth ride. In this second edition, I've removed any emphasis put on transitioning from ActionScript 2.0 to ActionScript 3.0.

I'll describe some features of the language here. This might help some of you who have several languages under your belts already and can benefit from a description of ActionScript 3.0. If you don't understand any of these terms, please don't fret! The rest of this Part exists to investigate these qualities of the language in depth.

ActionScript 3.0 can use both dynamic and strong typing, but the compiler, the language, this book, and most of the world want you to use it with strong typing. If you want to live in a dynamic world, you can do so by turning off strict mode in your compiler or development environment. ActionScript 3.0 is an object oriented language that makes heavy use of namespaces. It has facilities for reflection. It embodies some elements of functional programming.

Maybe it goes without saying, but ActionScript 3.0 is the primary language used in the Flash universe. However, it's not the only one.

Depending on the platform and runtime you're targeting — that is to say, where, on what device, and on what software you want your content to run — you might use another language that this book is not about. I'll get into this more once you look at the platforms and runtimes that exist in the Flash universe. But rest assured that ActionScript 3.0 is the way to go for the Flash Platform right now. Most other options are for older technology.

There's another language that's a big part of the Flash Platform, and that's MXML. MXML is a declarative XML language used to program Flex. The interesting thing is that MXML compiles into ActionScript 3.0 during building. It's also used interchangeably with ActionScript 3.0. I won't discuss MXML or Flex in this book, but because Flex is a superset of Flash, this book provides an excellent, maybe prerequisite, background for any Flex developer.

Mixed into ActionScript 3.0 are several microlanguages, tailor-made for solving specific kinds of problems more efficiently than the grammar of ActionScript 3.0 would allow. Technically, these are part of

the ActionScript 3.0 language specification, but when you use them you can tell instantly that another dialect is being spoken. These are E4X, a language for manipulating XML; regular expressions, a language for searching for and manipulating patterns of text; and XML itself, a way to store hierarchical information. Furthermore, there are closely associated languages, like Pixel Bender language, used to write Pixel Bender shaders. Although this language can't be written directly into ActionScript 3.0 code like E4X, it is necessary to use some of Flash Player's features. All of these languages are important parts of Flash Platform development and cannot be ignored by an ActionScript 3.0 developer.

There are two major points of confusion when speaking about the ActionScript 3.0 language. First is the fact that ActionScript 3.0 can change, and has changed, without changes to that little "3.0" number sitting in its title. It's a living language, and it's being developed even as you read this book. (Hello readers from the future! Have we learned how to speak to dolphins yet or what?) So how can Action-Script 3.0 change? Well, there are two ends that have to agree: the compiler has to make something out of your code, and the runtime has to be able to run what the compiler makes. But the fact is, all it takes is for the compiler to change to support a change in the language. Both can be modified at the same time, or the compiler only could change, to allow for alterations in the language. The compiler is the most important because it's the only part of the entire Flash Platform that sees actual ActionScript 3.0 source. This kind of change has already happened. The compiler shipped in Flash CS4, Flash Builder, and in newer builds of the Flex SDK supports syntax for parameterized types that you'll see in Chapter 9, "Vectors." This syntax, `TypeA.<TypeB>`, looks like utter rubbish to an older ActionScript 3.0 compiler like that in Flash CS3, yet it is (now) part of ActionScript 3.0. We simply have to be careful. The second point of confusion when speaking about ActionScript 3.0 is the difference between ActionScript 3.0 and the Flash Player API.

API

Where a language determines keywords (like `for`, `class`, and `is`), syntax (like where to put curly braces), and grammar (like how subexpressions are evaluated and what can and can't go on the left side of an assignment), it's really the *Application Programming Interface*, or API, that gets most of the work done. It's easy to confuse these two, so let's untangle them once and for all.

The language by itself can't do much of anything. Without an API it's little more than a glorified calculator. You can do operations like creating variables, assigning values, summing up things, and concatenating strings; you can even create classes and functions. That said, there's a whole lot to learn about the language itself, and you could in theory make it through the end of Part II, "Core ActionScript 3.0 Data Types," before using any of the Flash Player API, if you keep your eyes closed strategically. It's the runtime (Flash Player or AIR) that provides most of the exciting stuff the Flash Platform has to offer: graphics, sound, animation, networking, video, and so on. None of this is built into the language.

You can draw an analogy between programming languages and spoken or written languages. It's necessary to understand the grammar and pronunciation of a language, but that alone is not sufficient for communication. You need a rich vocabulary, and that's what the API provides.

If you don't mind skipping ahead to some topics discussed in Chapter 4, "Object Oriented Programming," an easy way to determine what's part of the core language and what's part of the API is to look at how it's namespaced. Any classes and functions in the default package are part of the language, like `Error`, `XML`, `int`, and `Number`. Anything in the `flash.*` package and its subpackages is part of the Flash Player API, like `flash.display.Sprite` and `flash.geom.Matrix3D`. The Flash Player API is a library of classes and functions that get real stuff done.

Each runtime you target when building a program has its own API associated with it. In this book, I'll only cover the Flash Player runtime (specifically only versions 9 and up) and the Flash Player API.

There is one more runtime I could potentially target: the AIR runtime. The good thing about the AIR API is that it's a superset of the Flash Player API. In other words, if you want to build AIR apps, everything you learn in this book will be applicable, and much will be necessary.

The runtime and the API it supports are fundamentally linked. Adobe engineers add new features into new versions of the Flash Player (the runtime) and simultaneously expose those features to programmers through additions to the Flash Player API. Changes to the API are far more frequent than changes to the language. That's why, in this book, I note when topics I discussed are particular to a certain version of the API.

Libraries

The Flash Player API is a staggeringly large collection of classes and methods that help you make interesting things happen on the screen, as the weight of this book attests to. But the brilliant thing about code is its extensibility. For every ability that the Flash Player API enables, there are dozens of ways that people have taken advantage of it to build something more complicated or make some task easier. And many of these people have been so kind as to share their hard-written code with the world (although a few do expect compensation).

When you build applications for the Flash platform, in addition to using the Flash Player API — a given — you can use any number of other libraries, leveraging their capabilities to pull off some impressive feats without breaking a sweat yourself. My favorite libraries include tweening libraries that let you animate things about the screen with a single line of code; physics engines that let you simulate collisions, friction, gravity, and other forces; 3D engines that let you present 3D objects and scenes; loading libraries that let you streamline the slightly annoying process of getting several types of assets loaded into a larger application; and data structure libraries that provide optimized storage for specific purposes.

A library is simply a collection of code that you can use. Sometimes you have the actual ActionScript 3.0 code that makes it up, and sometimes you use a precompiled binary (a SWC). In either case, you use libraries like you use the API. You create and access classes that they contain to get the job done. Libraries extend your programming toolbox with new, usually job-specific tools. In fact, the API is itself a library; what makes it special is that it's built into the runtime.

Because there are as many libraries as there are stars in the sky, I only mention them when there's a specific task that developers overwhelmingly use a library to accomplish, and in this case I'll just give a quick description of the library and let you know where you can find more information.

Compilers, Tools, and IDEs

There's one critical piece in this puzzle I haven't covered yet. ActionScript 3.0 code is text. You write it down in plaintext files. But to get from plaintext files to something you can actually run — in this case a SWF — you have to *compile* it. To put it simply, compiling translates the text of a program — the source code — into a simpler language that the runtime can run directly. We speak and understand human languages like English and French. Your computer speaks a certain instruction set that depends on what processor is inside it. It would be quite painful to program directly in the computer's language. (Some people come close by programming in assembly languages.) And it would be quite difficult and imprecise to have the computer interpret meanings from a regular, spoken language. So we have computer languages to bridge this gap — and compilers to translate. Unlike translating spoken languages, there can be no ambiguity in the compiler's translation. If the compiler doesn't understand the code you've written or believes it's incorrect, it fails to compile. You'll learn to recognize and deal with compiler errors in Chapter 5, "Validating Your Program."

So how do you use these compilers, and where do you obtain them? This all depends on the development environment you set up for yourself. In the Flash universe, most of the available versions of the ActionScript 3.0 compiler are integrated tightly with some tool.

The tools and Integrated Development Environments, or IDEs, you're likely to come across are

- Adobe Flash Builder
- Adobe Flash Professional
- FlashDevelop
- FDT

In addition, there are free and open-source versions of the Flex SDK available for download from Adobe. These toolchains are not IDEs, but they contain a compiler. All the tools in the previous list are either integrated with their own compiler or integrate tightly with a copy of the Flex SDK that you provide or that comes bundled with it. Because of the tight integration between these tools and the compiler, you may not even be aware of the compiler. For example, when you choose Publish or Test Movie in Flash, you invoke Flash's ActionScript compiler. With the default settings in Flash Builder, the compiler is invoked every time you save a file!

For every tool in this list but Flash, the compilers used are part of Adobe's Flex SDK, which comes in both free and open-source flavors. You can get the latest and greatest versions of these at no cost at `http://opensource.adobe.com/`. Specifically, these come with five major compilers: `mxmlc` to compile MXML and ActionScript 3.0; `asc` to compile ActionScript 3.0; `fsch`, a shell for repeated compilation; `compc` to compile SWCs instead of SWFs; and `adl` to compile and package AIR applications.

Source code is just text, so you can write it with any text editor you like. But because it's structured code, you can use tools to help you write this code faster and with fewer errors. IDEs, like Flash Builder, give you intelligent tools for writing ActionScript 3.0 code; searching it; discovering relationships between parts of the code; auto-completing your typing; exploring files; searching through projects; renaming classes, methods, and variables; and more. Possibly the most powerful feature of a good IDE, however, is its integration of an interactive debugger, which you can see how to use in Chapter 25, "Using the AVM2 Debugger and Profiler." Using an IDE makes a programmer's life much better, but you can always fall back on using a text editor to write ActionScript 3.0 source code and running `mxmlc` yourself. It's entirely up to you to decide what tools to use in your development environment. Personally, I recommend that you use Flash Builder if you're going to follow along with the book. I recommend against using Flash Professional for any serious ActionScript 3.0 programming.

Once you've compiled your ActionScript 3.0 program, you'll end up with a SWF file.

SWFs

A SWF file, or simply "a SWF," is an efficient, compressed binary file that can contain graphics, animation, text, bitmaps, sounds, video, and even arbitrary data. Most importantly for us, it also contains compiled ActionScript. The main purpose of a SWF is to get the stuff into the world (possibly across the great expanses of the internet, and onto the screens of your users) that we, as programmers, create. The end consumer of a SWF file is the Flash Player runtime.

I mentioned one other runtime: the AIR runtime. When compiling AIR apps, an `.air` file is generated, but even this is a package that contains SWFs for its executable ActionScript code.

Flex

I've tiptoed around Flex up until this point. Many newcomers to the Flash Platform are confused by Flex versus Flash, often with respect to the naming of certain products. Let's clear the air here.

Flex is two things. Primarily, it's a big, well-designed library for developing Rich Internet Applications, or RIAs. RIA is something of a vague buzzword encompassing programs that live on the internet and have some of the features once reserved for desktop applications: widgets like scrollbars to scroll, buttons and tabs to click, flippers to flip, and so on; multiple screens and transitions; display of tables and data. The Flex framework is a library that contains all these widgets, the ability to skin them, lots of code for easily connecting to web services, and more. As a library, it adds on to the capabilities of the Flash Player API, not replaces them. Furthermore, the end product of a program that uses Flex is a SWF, just like any other SWF, and it runs in Flash Player just like any other SWF. The only difference is that when you use the Flex framework, that SWF either contains or references the Flex framework library.

The second major component of Flex is a declarative XML language called MXML. This is a separate language from ActionScript 3.0, although when you compile a project with a Flex compiler, it is converted into ActionScript 3.0 code — and it coexists happily with ActionScript 3.0 code.

I won't cover either the Flex API or MXML in this book. However, it's important to know that Flex is still based in ActionScript and can be built targeting the same Flash Player runtimes. Because the Flex framework is a library built on top of ActionScript 3.0 and the Flash Player API, what you will learn in this book is indispensable for Flex development.

Note

A word about the use of the word "Flex" in product names: Flash Builder 4 is the successor to Flex Builder 3. Both of these tools can be equally well used to build either Flash or Flex applications, which the change of name is meant to emphasize. You'll also see the term "Flex" in the Flex SDK. The truth is that the Flex SDK bundles all you'll need to compile and package Flash, Flex, and AIR applications. ∎

In Short

Let's quickly put back together the programmer's experience with the Flash Platform, in a typical example. You open your IDE, Flash Builder. You start writing code in the ActionScript 3.0 language and use classes from the Flash Player API. You choose Run from the menu, and Flash Builder builds and runs your program, compiling all your ActionScript 3.0 code into a SWF and opening that SWF in Flash Player.

A User's Perspective

One of the biggest benefits of programming for the Flash Platform is that your content can be easily run in so many places for so many users. The two *runtimes*, Flash Player and AIR, are widely supported on multiple *platforms*.

Runtimes

A runtime is an environment in which a program executes. The runtime provides all the services necessary to do the things that the API promised were available. In the Flash Player API, you can create an instance of `Camera` to gain access to a connected webcam; the Flash Player runtime has to deal with the potentially complex task of finding the connected hardware and pulling a video stream from

it. In ActionScript 3.0 and the Flash Player API, you can programmatically draw graphics; it's up to the Flash Player runtime to render those graphics and work with the operating system to display them on-screen.

I've mentioned two runtimes: AIR and Flash Player. AIR, because it's a runtime made for desktop applications, contains additional capabilities, such as rendering web pages and spawning new windows in the user's operating system. Another runtime in the Flash universe, Flash Lite, is a somewhat dated environment for mobile and embedded devices. It doesn't support ActionScript 3.0, so I won't waste time on it. I'll only cover the Flash Player runtime here.

Platforms and Platform Independence

A platform is considered a combination of the hardware and operating system in use. More specifically, it is the instruction set of the CPU that matters to the platform, although these are mostly standardized. Users who want to run Flash content (stuff made with ActionScript 3.0 and the Flash Player API) must be able to run Flash Player on their platform.

At the time of writing, Flash Player 10 is supported on PCs with x86 processors running Windows 98 and up; Macs with PowerPC (G3, G4) or Intel processors running OS X 10.4 and up; PCs with x86 processors running several flavors of Linux including Red Hat, SUSE, and Ubuntu; and systems with x86 or SPARC processors running Solaris 10. Notably, at the time of writing, 64-bit processors are not supported, but commercially used 64-bit processors have no problem running in 32-bit mode, so I can still use Flash Player on my desktop's screaming Core i7 processor (which may be stone age by the time you read this). Also, at the time of writing, Flash Player 10.1 is planned to roll out on multiple mobile devices, including the latest Android OS and Palm webOS. In addition, there are other exotic Flash Player runtimes for platforms. For example, a PlayStation 3 with updated firmware has a Flash Player runtime on par with Flash Player 9.

In any case, that whole paragraph just means this: almost everyone can get access to the Flash Player runtime. One of the great things about the Flash Platform is that it's *platform independent.* It doesn't matter what kind of computer you use to compile your program: the resulting SWF file is completely indifferent to where it was born and where it's going. It's simply a binary file, and anyone who knows how to read and interpret it may do so, just like, say, a JPG image file. The other component of platform independence is that it doesn't matter what platform your user is on; users on different platforms are able to run the same SWF and see the same outcome.

This model of platform independence is just the same as Java's. Once Java code is compiled, it can be sent anywhere and run on any platform — that is, any platform where a Java Runtime Environment is available. Contrast this with traditional software development or game development, where a product is made for one platform and must be significantly reprogrammed or, at minimum, recompiled to be available on another platform. And considering that Flash content is widely distributed on the internet, the benefit of platform independence is clear. As the programmer, you don't want to have to make one SWF for every platform, and the users don't want to have to choose their platform every time they view a web page with Flash content.

The Flash Player Zoo

You've already seen that the Flash Player is available on different platforms. It also comes in several flavors and a plethora of versions.

There is a *standalone* Flash Player used to run SWF files from a user's computer. This can be bundled with a SWF to generate an executable that launches Flash Player and the bundled content at the same time. This is a Flash Player *projector.* Most common, there are the *plug-in* Flash Players that operate inside your browser to host Flash content in a web page. (On PCs these come in both ActiveX plug-in and Netscape plug-in varieties.)

Furthermore, all these flavors of Flash Player are available in debugger versions. You can learn more about the debugger versions in Chapter 24, "Errors and Exceptions."

Finally, there are lots of versions of Flash Player in existence. The major version of Flash Player is most important (for example, Flash Player 9 or Flash Player 10). The minor versions and revision numbers appear after the major version number, such as in Flash Player 10.0.22.87. In general, major versions introduce suites of new features, and minor revisions are mostly bug fixes and performance enhancements, although some feature changes creep in.

When you compile ActionScript 3.0 code, you can target a specific major version of Flash Player, because as I mentioned earlier, every version of Flash Player is tied to the Flash Player API for that version. All ActionScript 3.0 code is compiled to target Flash Player 9 at a minimum.

The good news is that you don't need to worry about the differences between all the flavors of Flash Player. The same SWF will work in Flash Player of the same version of every platform, no matter if it's the standalone version or the plug-in version, and no matter what browser the plug-in version is being hosted by.

In Short

Here you see a typical user's interaction with the Flash Platform. The user downloads or upgrades to Flash Player 10. She opens Firefox on her Mac and navigates to a page with Flash content. Behind the scenes, her browser downloads the SWF file from the internet and runs the plug-in version of Flash Player 10, and the Flash Player runs the content of the SWF. End result: she sees the Flash content hosted in the web page.

From ActionScript 2.0 to ActionScript 3.0

If you've programmed in ActionScript 2.0 before, ActionScript 3.0 has a whole lot of new features for you to explore. Here is an overview of the key new features. You may skip this section if you are coming to ActionScript 3.0 from a different language.

Display List

In ActionScript 2.0, there were three basic types of objects that could be displayed: movie clips, buttons, and text fields. These types didn't inherit from a common source, meaning polymorphism didn't work for these display types. Furthermore, instances of these display types always had a fixed, parent-child relationship with other instances. For example, to create a movie clip, you had to create that movie clip as a child of an existing movie clip. It was not possible to move a movie clip from one parent to another.

In ActionScript 3.0 there are many new display types. In addition to the familiar types such as movie clips, buttons, and text fields, you'll now find new types such as shapes, sprites, loaders, bitmaps, and more. All display types in ActionScript 3.0 inherit from `flash.display.DisplayObject`, allowing you to use them interchangeably in many cases. Furthermore, display objects in ActionScript 3.0 can be constructed independent of any other display object, and these objects can be associated as children of other display objects and even moved from one parent container to another. In other words, you can create a text field in ActionScript 3.0 simply by calling the constructor, and that text field will exist independent of any parent container object.

```
var text:TextField = new TextField();
```

11

You can then add the text field to a parent container at any time. The following example illustrates this with a display object called `container`, which could be a sprite or any other display object container type:

```
container.addChild(text);
```

Note

In the preceding example, `container` is used as a generic variable name that would presumably refer to an object created elsewhere in the code. ∎

The hierarchy of parent containers and their children is known as the *display list* in ActionScript 3.0.

Runtime Errors

ActionScript 3.0 provides many new runtime errors. This is an important new feature because it allows you to diagnose problems much more quickly. In ActionScript 2.0, when an error occurred at runtime, it would frequently occur silently, and it would be difficult for you as the developer to determine what the problem was. With improved runtime errors and error reporting in the debug player, it is now much easier to debug ActionScript 3.0 applications than it was with ActionScript 2.0.

Runtime Data Types

Strict typing in ActionScript 2.0 was only used by the compiler, not at runtime. At runtime, all ActionScript 2.0 types are dynamic. However, in ActionScript 3.0, strict typing is preserved at runtime as well. The advantage is that now runtime data mismatches are reported as errors, and application performance and memory management are improved as a result of preserved typing at runtime.

Method Closures

In ActionScript 3.0 all methods have proper method closures, which means that a reference to a method always includes the object from which the method was originally referenced. This is important for event handling, and it stands in stark contrast to method closures in ActionScript 2.0. In ActionScript 2.0, when you reference a method, the object from which the method is referenced does not persist. This causes problems, most notably when adding event listeners. In ActionScript 2.0, a delegate is often used as a solution. However, in ActionScript 3.0, delegates are not necessary.

Intrinsic Event Model

In ActionScript 3.0, the event model is built into the core language. Many native ActionScript classes, including all the display object types, inherit from the `flash.events.EventDispatcher` class. This means that there is one standard way to dispatch and handle events in ActionScript 3.0.

Regular Expressions

Regular expressions are a powerful way to find substrings that match patterns. ActionScript 3.0 includes an intrinsic `RegExp` class, which allows you to run regular expressions natively in Flash Player.

E4X

E4X is short for ECMAScript for XML, and it is a new way to work with XML data in ActionScript. Although you can still work with XML as you did in ActionScript 2.0 by traversing the DOM, E4X allows you to work with XML in a much more speedy and intuitive manner.

Summary

- AS3 is used to program interactive content for the Flash Platform.
- AS3 is object oriented and usually strongly typed, and it supports dynamic types.
- The Flash Platform encompasses languages, APIs, tools and IDEs, compilers, and the Flash Player and AIR runtimes.
- AS3 is the principal language used for Flash; MXML turns into AS3.
- Without the Flash Player API, you can't do much with AS3.
- The API is tightly tied to the runtime. This book is about ActionScript 3.0 and the Flash Player API.
- There are many tools for building Flash content. Most rely on the ActionScript compilers in the Flex SDK.
- You compile your ActionScript code to produce a SWF.
- SWFs are portable, compressed, and platform independent. They contain compiled code and other resources.
- SWFs are consumed by Flash Player.
- Flash Player comes on many platforms in many different flavors.
- All a user needs to run your Flash content is a compatible version of Flash Player.

ActionScript 3.0 Language Basics

S o you want to be an ActionScript coder? Great! This chapter will get you started with the basic syntax and structure of the language. If you have worked with other programming languages, some of the topics covered will be familiar territory for you. ActionScript's syntax is descended from JavaScript, Java, and C. One could call it "C-like," so if you've had experience with another C-like language, the syntax won't be difficult to pick up. Regardless of whether you know a similar language, you should read through this chapter closely so that you don't miss any of the nitty-gritty details.

The Bare Essentials

At its most basic, the grammar of ActionScript 3.0 is a series of statements and organizational structures. You write code in a plaintext file, optionally using spaces, tabs, and newlines as desired to improve readability.

ActionScript code is kept in ActionScript files, generally called *class files* because they generally contain one public class. These are plaintext files that you edit in your text editor or IDE, as discussed in Chapter 1, "Introducing ActionScript 3.0." The naming and organization of these files depend on their package structure, which you'll learn about in Chapter 4, "Object Oriented Programming."

Note
When using ActionScript 3.0 within Flash Professional, in addition to using class files, you can add code to a timeline. This is a separate way to edit code that doesn't require external files. I won't cover this technique here. ■

Usually, you'll write ActionScript code one line at a time. Each line is a single *statement*, and it's like a sentence in written or spoken language. After every sentence in written English, you use a period. After every statement in ActionScript,

you use a semicolon. In some cases, the compiler is lenient and can figure out that you're starting a new statement without the use of a semicolon, but it's good practice to always end your statements with a semicolon and a newline. Here's an example of three statements:

```
startTime = new Date();
var msElapsed:Number = endTime.time - startTime.time;
var hoursElapsed:Number = msElapsed / 1000 / 60 / 60;
```

Every statement here is like a sentence. You don't have to be able to understand this code just yet, but check out how each line does one thing, goes on its own line, and ends with a semicolon. The first line creates a new `Date` object. The second line calculates the number of milliseconds difference between this date and another date. The third line calculates the same figure in hours.

Expressions are a big part of most statements. All three lines in the example contain expressions. An expression is a piece of code that can be evaluated. For example, a simple expression is 1+1. It has multiple parts, but it can be turned into one value — 2, of course — through a simple evaluation. Expressions can contain combinations of operators (introduced in the section "Connecting You to an Operator"), assignments, and function calls (introduced in Chapter 3, "Methods and Functions"). To be evaluated, the operators are applied to their operands, the values are assigned, and the functions are executed. The results of these actions are substituted over and over until the expression is a single value.

In ActionScript 3.0, code is also organized into blocks. A block of code is one or more statements enclosed in curly braces. Following is a useless block of code:

```
{
  1 + 1;
  "I am " + "a banana!";
  var x:Number = Math.round(Math.PI);
}
```

Blocks group together related code and create local scopes for the creation of variables. The bodies of functions, classes, and packages are all contained in blocks. Usually you don't just create blocks by themselves as in the previous example (although you can), but the block is a natural part of another construct like a function or a loop.

The term *whitespace* refers to any spacing put in your code. This can take the form of spaces, tabs, or newlines. Extra whitespace is utterly ignored by the compiler, and only a bare minimum is required for the compiler to figure out what you mean, so you can include as little or as much whitespace as you like. Style conventions tell you how you should space out your code, though no one set of conventions is "correct." You can follow the conventions I use in examples, a different conventional style, or even make up your own style. What's most important is that you stay consistent. The following two statements are equivalent:

```
phoneBook.callFriends(["Alice","Bob","Charlie"]);

phoneBook . callFriends(
  [
    "Alice",
    "Bob",
    "Charlie"
  ]
);
```

You won't be able to see what a typical class file looks like in its entirety until you've learned about packages and classes in Chapter 4, but now you know that all ActionScript code is constructed of statements, expressions, and blocks of code. Let's learn how to make these useful.

Using Variables

Not to scare you away from programming if you happen to hate math, but programming is just like algebra: its power and generality are drawn from using variables to represent numbers. A formula like $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ that computes the Euclidean distance between two points (x_1, y_1) and (x_2, y_2) is powerful because it applies to *any* two points you can think of. To calculate it, you simply substitute your coordinates in the proper place and do the arithmetic. ActionScript 3.0 is built entirely around the use of variables just like the *x* and *y* variables in an algebraic equation.

A variable is a representation of a number, string of characters, or some other data that can change values the way the *x* variable in algebra represents a number that may have any value. You might think of variables as containers used to hold pieces of information. In ActionScript, variables contain chunks of information known as objects for the time your program is running. They allow your program to hold on to that information and operate on it. Variables are used in every aspect of programming. You may also hear variables referred to as *properties* or *fields* when they are part of a class.

Anatomy of a Variable Declaration

Defining your own variables is easy. Let's take a look at a typical variable declaration and step through its different parts in the list that follows:

```
var food:String;
food = "pizza";
```

- `var` — All variables are defined by using the `var` keyword followed by the name of the variable. This is different from previous versions of ActionScript where the `var` keyword was optional.

- `food` — This is the name of the variable. It can be any word or string of letters and numbers beginning with a letter or an underscore. It can't have any spaces in it, and it can't be a reserved word. By convention, I recommend naming your variables using descriptive words in camel case and starting with a lowercase letter. For example, `pizzaToppings`, `selfDestructCancelTime`, and `licksToTheCenter`. Go ahead and be specific; other users of your code will thank you. You might even thank yourself later!

- `:String` — This defines the data type or the type of information that the variable can hold. The word `String` could be replaced with any class or interface name. I'll talk more about data types in the later section "Introducing the Data Types."

- `food = "pizza";` — This statement sets the value of `food` to the word `"pizza"` using the assignment operator (=). The quotes around the word `pizza` signify that this is text. (They signify a string literal, which you'll learn about in Chapter 6, "Text, Strings, and Characters.")

The first line is called a *declaration*. The name and type of the variable are declared. The second line shows *assignment*, where a value is assigned to the newly created variable. You can also declare and assign variables a value in a single line:

```
var drink:String = "Root beer";
```

You can even declare several variables on a single line, separated by commas:

```
var breakfast:String, lunch:String, dinner:String;
```

Or you can declare and assign multiple variables at once:

```
var breakfastTime:Number = 8, secondBreakfastTime:Number = 10,
    elevensesTime:Number = 11, luncheonTime:Number = 12.5,
    afternoonTeaTime:Number = 3, dinnertime:Number = 5,
    supperTime:Number = 7;
```

After you've defined a variable, you can use it in an appropriate context. Its value will be substituted when the code is actually executed. For example:

```
trace("My favorite food is " + food);
// displays "My favorite food is pizza"
```

Here you typed `food`, but what came out is `pizza`. That's because `food` is a variable name, and at runtime its current value `"pizza"` was substituted.

Note

You'll see the `trace()` function used continually throughout the book. This simply prints out what you tell it to. How it prints out depends on how you run the code. In Flash Builder or Flex Builder, you should see this output in the console when you debug the project. In Flash Professional, you should see it in the Output window when you test the movie. When running the examples from their provided URLs, the output should appear on-screen. However it prints out, `trace()` is used in examples to illustrate what is going on. In most examples, I've put what should be the output next to the `trace()` statement in a comment. This way, you can read along without running the code. Read on for more information about functions and comments. ■

Constants in This Changing World

ActionScript 3.0 has a special kind of variable called a *constant*. Constants define values that don't change during the course of your program. This is great for values that either never change by nature or that you want to protect from change. For example, if you wanted to store the boiling point of water in Fahrenheit, and you assume that nobody's going to take your program to high altitudes, you could write the following:

```
const BOILING_POINT:int = 212;
```

As you can see, this looks a lot like a variable declaration but with the `var` keyword replaced by `const`. Constant statements can be defined only at the beginning of a class with the other properties or in the constructor for a class. Attempting to assign a new value to a constant anywhere else will result in an error.

Constants are used extensively in the Flash Player API, especially in the `flash.errors` and `flash.events` packages. By convention, constant names are written in all caps with words separated by underscores.

Taking It Literally

Aside from variables and values returned from functions, you will use *literals* in your code. The term literal refers to any value that is explicitly included in the code at the time it's compiled. It offers a

convenience when dealing with complex data types. Following are the different types of literals you can include directly in your code:

- Numeric values like `98.6`, `-100`, `0xff`, and `6.02e23`

- Boolean values `true` and `false`

- Strings written with single or double quotes, like `"Lorem ipsum"`

- The empty value `null`, the undefined value `undefined`, and the non-number `NaN`

- Array literals that use square brackets, such as `["Monday", "Wednesday", "Friday"]`

- Generic objects defined using the curly bracket syntax such as `{name:"Lovefoxx", likesToDance:true}`

- Regular expressions enclosed in forward slashes, as in `/href="(.+?)"/`

- Elements written in XML such as `<menu><item id="bacon" label="Press to Receive Bacon"/></menu>`

You can find much more about these literals in the chapters on their respective data types.

Tip

I recommend that you avoid using literals frequently in your code, because they can be difficult to track and change. If you use the same literal a few times, define a variable or constant with the literal value and use that instead!

If you wanted to change this program from Fahrenheit to Celsius, you'd have to change 212 in multiple places:

```
if (temp > 212) {
  flame.level = 0;
  temp = 212;
} else {
  flame.level = 10;
  while (temp < 212) {
    temp = temp + 1;
  }
  flame.level = 2;
}
```

However, use a constant, and you only have to change one value:

```
const BOILING_POINT = 100;
if (temp > BOILING_POINT) {
  flame.level = 0;
  temp = BOILING_POINT;
} else {
  flame.level = 10;
  while (temp < BOILING_POINT) {
    temp = temp + 1;
  }
  flame.level = 2;
}
```

As a side effect, your code becomes more self-explanatory. You could even use `FLAME_MAX` instead of 10 and `FLAME_OFF` instead of 0. ∎

Commenting Your Code

Comments are sections of text that appear in your code but aren't part of the program. They are usually plaintext and not ActionScript, and the compiler ignores their existence entirely — passing right over them. Comments are used to add notes to yourself or others who may work on your code. Although some programmers shy away from using comments because it seems tedious or frivolous, leaving comments is a good habit to get into. Comments can provide a quick explanation for what your code is doing, saving you the time and confusion of trying to interpret the code, which of course makes them most valuable for the most complicated parts of your code. When you are working with others, comments become a helpful tool for describing your intention to other programmers. Or they can be useful for your own benefit if you haven't looked at a piece of code in a long time. Some specific uses for comments are presented later in this section.

Types of Comments

There are two types of comments in ActionScript 3.0: single line and block comments. Because comments are purely for your benefit, they are utterly interchangeable. Use whichever suits the task at hand. Typically, single-line comments are used for shorter comments, and block comments are used for longer chunks of text.

Single-Line Comment

You start a single-line comment by typing two forward slashes (//). Everything following the slashes on the line is treated as a comment, which is to say, totally ignored by the compiler. Single-line comments are useful for adding a note to a line of code:

```
var duration:Number = 400; //measured in milliseconds
var n:Number = (2*Math.random() - 1) * 100 //between -100 and 100
```

Block Comments

Block comments include all text between a starting and an ending marker. Use /* to start your comment block and */ to end it. The compiler will ignore all text between the two markers.

```
/*
   This is an example
   of a multi-line
   block comment.
*/
```

XML Comments

A third type of comment you might encounter is an XML comment. This is a block comment that is used exclusively within XML code (including MXML). XML comments start with <!-- and end with -->. The characters -- must not be included anywhere between the start and end markers.

```
public var xml:XML = <colors>
    <color name="chartreuse" hex="0xDFFF00"/>
    <color name="mauve" hex="0xE0B0FF"/>
    <color name="cerulean" hex="0x007BA7"/>
    <color name="Parcheesi" hex="0xF4A460"/> <!-- not sure this is a color -->
  </colors>;
```

Javadoc Comments

Javadoc (or ASdoc) comments are specially formatted comments that are ignored by the compiler but heeded by a special tool that creates documentation. You'll know you're seeing javadoc comments because of the delimiters, a /** start tag and a */ end tag. Also, certain directives inside the comment, like @param, @return, @deprecated, and so on, are included in the syntax to help create documentation. Typically, you'll see javadoc comments before method signatures and at the top of classes.

```
/**
 * Calculates time until birthdays.
 * @param   your birthday
 * @return  the number of days until your birthday, rounded up
 */
public function daysUntilBirthday(bday:Date):int {...
```

See the documentation for your autodoc tool for more information about javadoc-style comments. More information is available at http://bit.ly/asdoc-help.

When to Use Comments

As you're starting out, it's not a bad idea to add comments anywhere that your code might not be completely self-explanatory, especially with algorithms that may be unfamiliar to you or other people. Comments can be great for adding notes to remind yourself of something. A lot of people tag their comments with keywords like TODO or XXX or FIXME, as in:

```
// Calculates the area of a square.
// TODO add a separate height and width to support rectangles.
public function calculateArea(width:Number):Number {
    return width * width;
}
```

Comments can also be used to *comment out* a line of code, which allows you to keep a line of code in the file but ignore it, so that you can put it back later if taking it out turns out to be a mistake. The following code:

```
trace("hello world");
```

will display hello world, whereas the following will do nothing:

```
//trace("hello world");
```

Tip

As a matter of style, don't let your code get too cluttered with ancient commented-out lines. Purge them when you're sure you don't need them anymore. If you use version control software like SVN, you can always go back in time to see code even after it's been deleted from the source files. ∎

Self-Commenting Code

Some developers, myself included, would argue that the best kind of comments is no comments; writing code so descriptive and obvious that comments are rarely if ever needed. This is called *self-commenting code*. Write self-commenting code by using descriptive words for all variable names

and method names, using strongly typed and well-defined variables, and encapsulating complex functionality into easy-to-describe functions. This leads to less reliance on commenting. Self-commented code should ideally be understandable even by a non-programmer.

Introducing Scope

All variables and functions defined in ActionScript exist in a certain scope. This scope is defined as the zone in which the object can be accessed. A variable or function is *in scope* when you're writing code that can access it; if you type a variable or function name that's out of scope, ActionScript won't recognize it.

In the United States, the national government deals with issues affecting the entire country, such as tax law; local governments deal with local issues, such as funding for a certain town's school. National governments aren't equipped to handle local issues and vice versa. In most cases the national laws are used except when overridden by a local law.

Scope works in much the same way. Variables operate in a particular scope depending on how and where they are defined. Generally speaking, the more broadly defined variables are used except when replaced by local versions. Let's look at the different types of scope that you'll be dealing with.

Scope can be either *global* or *local*. Objects in the global scope can be accessed from anywhere in the code, whereas objects in local scopes can be accessed only through the object where they are defined. In all code in this book, scope is controlled by class structures, their access control modifiers, and namespaces. This level of control over scope is more fully described in Chapter 4.

ActionScript 3.0 doesn't truly have a global scope that you can access. Everything is neatly organized into packages and smaller structures. However, many classes and functions are placed in the default package, which is always in scope.

Local scope is a bit more complicated. There are several layers of localized access depending on how the object is defined. In general, the levels of scope are associated with the nested blocks of code. If you are writing code inside a block (remember these are groups of statements inside curly braces) and you reference a variable, ActionScript 3.0 looks for that variable declared inside the block, then (if it's not found) in the block enclosing that, and so on. This general principle is modified by access control modifiers. I'll cover packages, classes, and methods in Chapter 4, so I'll revisit scope in more detail then.

Caution

In reality, this guiding principle is not implemented precisely. ActionScript 3.0 doesn't have true block-level scope. But it does have levels of scope associated with the package block, class block, method block, and function blocks. Other blocks, like loop bodies, don't actually have their own scope. You can declare variables in them, but when they are compiled, the code acts as if all variables in a function, including these, were declared at the top of the function. This is known as *variable hoisting*. ∎

I introduce scope here to explain the fact that when you declare a variable, it has a certain sphere of influence. As you build functions, classes, and bigger structures, this will become more important.

Introducing the Data Types

ActionScript 3.0 is a *typed language*. That means that every object (variable) is assigned a data type, which determines what kind of variable it is and what data it stores. A variable's type tells the compiler and Flash Player what type of value to expect. You use types both when declaring variables and when passing them around your program, as you'll see with functions.

Declaring Types

To declare the data type of a variable, follow the name of the variable with a colon and its type. You can think of this like the convention of written English like "Roger Braunstein: Purveyor of Pugilism." Once you declare the type of a variable, the compiler works to ensure that only proper types of data are stored in the variable.

```
var x:Number;
var name:String = "Roger";

x = 42;    // No problem
x = -13;   // No problem
x = 3.141; // No problem
x = "foo"; // Throws compile time error!
x = name;  // Throws compile time error!
```

Because x is defined as a `Number`, it can't be assigned a `String` value like `"foo"`. The compiler checks all the incoming data types to make sure they're compatible. If a problem is encountered, the compiler produces an error.

Errors sound bad, but compiler errors are your friends. Imagine what would have happened if there were no error. The value of x would suddenly be a `String`. Then the next time you tried to use it as a `Number`, the results wouldn't make sense (`"foo"` * 8.5 = *???*). Keeping data types intact helps your code to be predictable and stable.

Using Untyped Variables

Occasionally, you may need to store a value in a variable without knowing what type of data it will hold. For this, AS3 allows the use of a wildcard (*) data type. The * denotes that a data type is unknown until runtime (dynamic) or will accept more than one type of data (for example, both strings and numbers). The wildcard is useful when creating functions that behave differently depending on the data type of its parameters.

You should exercise caution when using these types because type checking at both compile time and runtime will not be available for these variables, and your results may be unpredictable.

Connecting You to an Operator

Operators are built-in, often symbolic entities that perform a specific task on one or more values. Big subsets of these operators are logical and arithmetical, so you will probably recognize and intuitively understand many of them. Some of the operators are symbols (such as +), and some are words (such as `return`). Operators can come before, between, or after their operands. (These are called *prefix*, *infix*, and *postfix* operators, if you want to get technical.)

Unary vs. Binary Operators

Some operations are applied to a single argument, such as the increment operator (++); these are unary operators. Others operate on two operands, such as the plus operator (+); these are called binary operators. There is also a ternary operator, which takes three arguments. It's important to provide the correct number of arguments when using operators, which isn't difficult because they're fairly self-explanatory. (You can't add only one value.)

```
var counter:int = 0; //counter is 0
counter++; //now it's 1. ++ takes one argument, counter
counter = counter + 5; //now 6. + takes two arguments, counter and 5
```

Order of Operations

Operators follow a certain order of operations. That is, operators with a higher precedence are processed before ones of lower precedence. For one, this order of operations preserves the order of operations in written arithmetic. In general, more complex operations are executed first, followed by simpler operations.

ActionScript executes operations in more or less the following order:

1. Expressions contained within parentheses are executed starting with the most deeply nested, and working outward.
2. The results of functions are evaluated, assuming all their arguments are evaluated.
3. The multiplication, division, and modulus operators (*, /, %) are applied from left to right.
4. The addition and subtraction operators (+, -) are applied from left to right.

As rule 1 implies, you can use parentheses to group expressions into subexpressions that will be evaluated before moving on to other operations. For example:

```
trace(3 + 4 * 5); // Displays : 23
trace((3 + 4) * 5); // Displays: 35
```

Use parentheses to create your own order of operations.

Some Commonly Used Operators

Here are some operators that you're likely to see. This is just a small sample of the operators available to you in ActionScript 3.0. Some other operators are more specific to certain types and are covered in their appropriate chapters.

Assignment (=)

Undoubtedly the most important operator, the assignment operator sets the value of the variable on the left side to the expression on the right side. The left side must be a single valid variable or property; you can't have expressions on the left side to dynamically choose what variable is being assigned to. The right side can be any expression that ActionScript 3.0 can evaluate.

Arithmetic (+, −, *, /)

These should need no explanation, perhaps except that multiplication is written as a * instead of ×, and division is written as / instead of ÷. Trust me — this way is much easier to type.

Modulo (%)

The modulo operator returns the remainder after a division operation. It divides the first number by the second and returns the leftover portion only.

```
0 % 3 // 0
1 % 3 // 1
2 % 3 // 2
3 % 3 // 0
4 % 3 // 1
5 % 3 // 2
6 % 3 // 0
...
```

Increment (++) and Decrement (−−)

These operators add or subtract 1 from the number they're used on. These are unary operators, so they require only one argument:

```
var x:Number = 5; //x=5
x++; //x=6
x--; //x=5
++x; //x=6
```

When the increment or decrement operator is placed before the operand, ActionScript processes the operation with a higher precedence.

```
var x:Number = 5;
trace(x++); //prints 5, then increments x to 6
trace(++x); //increments x to 7, then prints 7
```

Compound Assignment Operators (+=, -=, *=, /=, and %=)

The compound assignment operators provide a shorthand solution to performing arithmetic on a variable and storing the result in the same variable. For instance, += adds the left-hand variable to the right-hand expression, storing the result in the left-hand variable.

```
x += 1;
```

is the same as writing

```
x = x + 1;
```

The following code shows more examples of these compound operators.

```
var x:Number = 0;

x += 5; // x = 5
x -= 3; // x = 2
x *= 3; // x = 6
x /= 2; // x = 3
x %= 3; // x = 0
```

Comma Operator (,)

The comma operator joins several subexpressions. The whole expression takes the value of the last subexpression in the "list." You don't see this one used frequently, but it can be helpful.

```
var a:Number = 50;
var b:Number = (a *= 2, 50 + 50);
trace(a, b); //100 100
```

This is a tricky one. The snippet demonstrates that even though only the last expression separated by commas is returned, all of them are evaluated. The variable b is assigned 100 after evaluating the second subexpression 50+50, but you can tell by the fact that a becomes 100 that a*=2 was evaluated, too.

I'll introduce more of the operators in subsequent parts of this chapter. A comprehensive chart of all operators and their uses can be found in your trusty AS3LR.

Note

As mentioned in the Introduction, the ActionScript 3.0 Language Reference (AS3LR) or ActionScript 3.0 Language and Component Reference (AS3LCR) is the definitive source of information on all parts of the Action-Script 3.0 language. This document is available free online and is bundled with all of Adobe's ActionScript 3.0 development tools. If you'd like to see a full list of all the operators, refer to the "Language Elements" section of AS3LR. Find it at `http://bit.ly/as3lr`. ∎

Making Logical Choices with Conditionals

What good would it be if your programs and functions executed the same way every time you ran them? Essentially, the element of choice over how to act under different circumstances would be lost. That's where *conditionals*, or decision-making points in your code, come into play. Conditionals, such as the if statement, enable you to evaluate the truth of a logical expression and react differently in different scenarios.

All conditional statements deal with *Boolean* values. Booleans are simple logical values that can only be one of the values true or false. Any logical expression can be evaluated down to either true or false and used with conditional operators.

if Statements

The most commonly used conditional is the if statement. An if statement allows you to execute a piece of code only *if* a certain condition is met: if this is true, then do that.

```
if (logical expression) {
    // code to execute if the expression is true
}
```

An if statement is a whole statement, but unlike other statements, you usually don't end it with a semicolon after the block closes. In general, you don't use a semicolon after the end of a block (although adding one won't do any harm).

Here's an example of an `if` statement. It checks to see if the value of `weather` is equal to the string `"rain"` and if so it calls the function `bringUmbrella()`, a hypothetical function that brings along your umbrella. Don't worry about the function syntax: you'll learn about functions in detail in Chapter 3.

```
if (weather == "rain") {
    bringUmbrella();
}
```

Note

{} are optional for the body of an `if` statement that is only one line, but I recommend that you always use brackets for the sake of consistency and readability. In examples, you may catch me omitting brackets if I can fit the block on the same line as the `if` statement. I'm just trying to save you some pages. ∎

Equality (==)

In the preceding example, the `if` keyword is followed by the logical expression to evaluate in parentheses. Notice that there are two equal signs and not just one. This double equal sign is known as the equality operator (==). It checks to see whether the value on the left is equal to the value on the right. This is quite different from the assignment operator (=), which sets the variable on the left to the value on the right. The double equal sign is a logical comparison operator, which means it evaluates to a single Boolean result: either `true` or `false`. In this case, you're checking to see if the value of the variable `weather` is equal to the string `"rain"`.

Caution

Using the single equal sign instead of a double equal sign in a comparison statement can cause confusing and unexpected results. This typo happens to even the most experienced developers when they've been programming too late into the night (or next day). The compiler can sometimes identify when you're accidentally using = instead of ==, giving you a warning. Some developers get into the habit of putting the literal on the left when comparing to a value, as shown. Because it's not legal to assign a value to a literal, if you are in this habit and accidentally use = when you mean ==, the compiler will catch it as an invalid assignment.

```
if (0 == counter) //putting the literal on the left
if (0 = counter) //accidentally typing this yields an error
if (counter = 0) //this is perfectly legal though wrong
```

After the logical expression, you have the block of code that will be executed if the expression is found to be true. In this case, you're calling a function called `bringUmbrella()`. If weather is rainy, this block of code will be run. If the value of `weather` is anything else, the comparison evaluates to `false` ("no, weather does not equal rainy") and the body of the `if` block will not be run. Execution of code will continue after the `if` block.

Testing Other Comparisons

Logical expressions can evaluate more than just equality. ActionScript includes many other types of comparison operators and logical operators, which you can use alone or combine into more complex types of comparisons.

Greater Than (>) and Less Than (<)

Besides checking for equality, you can compare whether a value is less than or greater than another number by using the corresponding operators, < and >. The following code says "If the rainfall is more than 0, the weather is rainy."

```
if (rainfall > 0) { weather = "rain"; }
```

You can also use less than or equal to (<=) and greater than or equal to (>=), as follows:

```
var x:int = 0;

var a:int = -1;
var b:int = 0;
var c:int = 1;

trace(x <= a); // Displays: false
trace(x >= a); // Displays: true
trace(x <= b); // Displays: true
trace(x >= b); // Displays: true
trace(x <= c); // Displays: true
trace(x >= c); // Displays: false
```

Not Equal to (!=)

Adding an exclamation point before a bit of logic will evaluate the opposite of the expression. This is known as the not operator (!). Similarly, ActionScript includes a not equal to operator (!=) which, as you might imagine, checks to see if the value on the left is not equal to the value on the right:

```
if (weather != "sun") { bringUmbrella(); }
```

This statement brings an umbrella if it is *not* sunny outside. Another way of writing this would be to use the not operator to negate an equality:

```
if (!(weather == "sun")) { bringUmbrella(); }
```

This line of code functions identically to the one earlier. Because the not operator binds very tightly (it has a high precedence in the order of operations), you have to use parentheses. Otherwise, the expression !weather == "sun" evaluates whether *not-weather* is equal to "sun."

And (&&) and Or (||) Operators

If you want to act on a combination of conditions, you might try nesting or duplicating your if statements together like this:

```
if (weather == "rain")  { bringUmbrella(); }
if (weather == "sleet") { bringUmbrella(); }
if (weather == "snow")  { bringUmbrella(); }
```

```
if (weather != "rain")  {
    if (weather != "sleet") {
        if (weather != "snow") {
            leaveUmbrella(); }
    }
}
```

As you can see, this takes a lot of effort to write. Luckily, ActionScript uses two logical operators that allow you to combine multiple conditions into a single if statement. They are and (&&) and or (||). Using the and operator, both operands must be true for the expression to be true. You might say "If a and b are true, then do something."

Caution

With these operators, like the equality operator, you have to make sure to use two of the characters. The & and | operators exist for bit math, covered in Chapter 13, "Binary Data and ByteArrays." Remember to use && and || for logical combinations. ∎

Using the or operator, the expression is true when either (or both) of the operands are true. It basically says, "If a or b is true, do something."

Let's rewrite the preceding examples using and and or.

```
if (weather == "rain" || weather == "sleet" || weather == "snow") {
    bringUmbrella();
}

if (weather != "rain" && weather != "sleet" && weather != "snow") {
    leaveUmbrella();
}
```

Checking for Null Values

In ActionScript 3.0, null is a special value (non-value, really) that signifies *nothing*; it's sometimes used for empty variables that haven't been assigned anything yet, values that can't be represented and values that can't be determined. When you're working with objects that could be null, it's important to check whether the value is null before attempting to access methods or properties of the object. Failure to do so can result in a runtime error. Checking for null is simple:

```
if (weather != null) { checkWeather(); }
```

If the weather is not null, the checkWeather() function is called. You can also use the following code to achieve the same result.

```
if (weather) { checkWeather(); }
```

The value for weather will be converted into its Boolean equivalent before being evaluated. There are special rules for coercing types into other types implicitly. With objects in general, any non-null value is coerced to true, whereas null or undefined values are coerced to false. This way of checking is easier to write but is less reliable because it can cause different results depending on the data type of the object you're checking. Strings and Numbers have special rules. Because weather is presumably a String, it goes by String coercion rules, in which the empty string "" is also considered to be false.

if..else

With the `else` keyword, you can add an additional block of code to an `if` statement that will be executed if the condition is *not* met. Naturally, with its addition the conditional is called an `if..else` statement. The example that follows could be read as, "If it's raining, bring an umbrella; otherwise, leave the umbrella behind."

```
if (weather == "rain") {
   bringUmbrella();
} else {
   leaveUmbrella();
}
```

Furthermore, right after `else` you can add another `if` statement that will be checked only if the first `if` statement is false. The statement that follows says, "If it's raining, bring an umbrella; otherwise, if it's snowing, bring a coat; otherwise, if it's sunny out, bring sunscreen."

```
if (weather == "rain") {
   bringUmbrella();
} else if (weather == "snow"){
   bringCoat();
} else if (weather == "sun") {
   bringSunscreen();
}
```

Only one of these branches will ever be run. Once any of the `if` clauses evaluates to `true`, the code in the associated block is run, and control of the program passes to whatever follows the whole `if..else` structure. You can keep chaining on `else if`s to your heart's desire. And you can end the whole shebang with one final `else`, which executes if none of the previous if branches was chosen.

Note
The `else if` construct is not a special kind of statement. It's simply putting another `if` statement inside the `else` block of the previous `if` statement. Omitting the braces in this case keeps things tidy and lets you read the code as "else if...," even if it's really "else: if...." ∎

switch

If you need to test the condition of something against several possible outcomes, as in the last example when you were testing for rain, snow, and sun, it might be best to use a `switch` statement. A *switch* is just like a giant `if..else if` chain where each `if` tests the same expression (in this example, the weather). You can check for any number of different *cases* or possible outcomes for the expression you're evaluating and even define a *default* behavior in case none of the cases match.

Notice that `break` statements are used after each case. The `break` statement tells the conditional statement or loop to cancel the rest of its operation without going any further. Unlike the `if` statement, which executes code in a block enclosed with curly braces (`{}`), when working with switches, you must include `break` after each `case` to prevent the code from continuing to the next case.

```
switch (weather) {
  case "rain":
    bringUmbrella();
    break;
  case "snow":
    bringCoat();
    break;
  case "sun:
    bringSunscreen();
    break;
  default:
    stayInside();
}
```

This is equivalent to writing:

```
if (weather == "rain") {
  bringUmbrella();
} else if (weather == "snow") {
  bringCoat();
} else if (weather == "sun") {
   bringSunscreen();
} else {
  stayInside();
}
```

It's not required, but you can use a `default` block to define a case that executes if none of the other cases in the `switch` statement match.

Tip

You can intentionally omit the `break` statement after a case. This allows the subsequent cases to run in addition to the current one, if they also match the condition. For example, you could replace

```
} else if (weather == "thunder" || weather == "lightning") {
stayInside();
}
```

with

```
case "thunder":
case "lightning":
  stayInside();
  break;
```

The Conditional Operator

The final method that you can use to branch depending on a Boolean expression is the *conditional operator*, sometimes called the ternary operator because it is the only operator to take three operands (`?:`). The conditional operator behaves similarly to the basic `if` statement.

```
(logical expression)? value if true : value if false
```

31

The conditional operator takes three expressions as operands. The first is a logical expression to evaluate for truth, much like the expression passed into an `if` statement. This can be a comparison, a variable, a function call, or any expression that can evaluate to a Boolean value. It may be required to couch the expression in parentheses to ensure it's interpreted in the correct order; you may choose as a matter of style to always enclose the expression in parentheses.

The logical expression is followed by a question mark. Immediately after the question mark is the expression that will be evaluated and returned if the logical statement is found to be true. This is followed with a colon, and finally, the expression to be returned if the first value was found to be false.

Here's a practical example. This time, you check for hail:

```
(weather == "hail")? bringMotorcycleHelmet() : bringStrawHat();
```

This is basically the same as writing:

```
if (weather == "hail") {
  bringMotorcycleHelmet();
} else {
  bringStrawHat();
}
```

The thing that makes the conditional operator unique is that the entire thing evaluates to the result of the correct branch, rather than simply executing it like an `if..then..else` statement. So if it's hailing, not only will `bringMotorcycleHelmet()` run, but the entire conditional expression will be replaced with whatever value `bringMotorcycleHelmet()` returns. So you can use conditional expressions within other expressions. The following:

```
hatType = (weather == "hail")? "helmet" : "straw";
```

sets `hatType` to `helmet` or `straw` depending on the value of `weather`.

One common usage of this operator is to set a default value for values that may or may not be defined:

```
weather = weather? weather : "partly cloudy";
```

The preceding code might look a bit confusing, but what it's doing is simple. It's checking to see if `weather` is defined and, if not, assigning the value of `"partly cloudy"`. Because most objects in ActionScript evaluate to `true` if they have a value and `false` if they are `null` or `undefined`, the expression `weather?` will evaluate to `true` if there is a definition and `false` otherwise. One of the two following expressions will be returned and assigned to the variable resulting in either:

```
weather = weather
```

which causes no changes, or assigning the default value of `"partly cloudy"`:

```
weather = "partly cloudy"
```

Caution

Although conditional statements are much more compact to write, they can be more difficult to read and understand quickly than `if` statements, especially when the conditional expression is complex. Use them when you need to switch between values within an expression. ∎

Repeating Actions Using Loops

Loops are control structures that allow you to repeat a block of code multiple times, often while progressing through some set of data. Just as conditional statements make your programs interesting by providing points of variation, loops make your programs interesting by allowing you to perform operations that you wouldn't be able to do by hand. After all, isn't the computer supposed to be doing the work?

The uses for loops are endless. For example, you might want to read a list of 100 items, apply a calculation to every graphic on the screen, search through an array for a particular value, initialize every object in a list, or just write every letter from A to Z.

Using for Loops

There are a few different types of looping structures in ActionScript 3.0. Let's start with the classic, the original, the `for` loop. If you've used any C-like language, you are probably familiar with this structure.

A common `for` loop looks something like this:

```
for (var i:int = 1; i <= 100; i++) {
   // do some action
}
```

If you're not familiar with loops, however, that code might look a little bit scary, but really, it's not so bad. You can read this as follows. "Start `i` at 1. Keep going as long as `i` is 100 or less. Count up `i` 1 at a time." Or, "For every integer `i` from 1 up to and including 100, do some action." Or, "Repeat some action 100 times."

Let's break it into sections:

- `for` — All `for` loops start with this keyword.
- `var i:int = 1;` — The loop initializer that runs once as the loop begins. You're creating a new integer called `i` and setting it to 1. This variable, `i`, will be the number that counts up during each loop. Loop variables are frequently named `i`, sometimes proceeding to `j`, `k`, and so on, when there are multiple nested loops. The letter `i` stands for *iteration* or *index*. The type of the loop variable can also be a `uint` or a `Number`.
- `i <= 100;` — The conditional. Defines the condition under which the loop may continue running. This is like an `if` statement that asks, "If `i` is less than or equal to 100, keep on going." The number 100 is arbitrary and should be replaced with whatever your upper limit is.
- `i++` — The counter. Tells the `for` loop what to do after each iteration of the loop. Usually counts up or down on the loop variable. Remember that the ++ operator adds one to the argument.

33

- `// do some action` — The loop body. All code inside this block gets executed during each iteration. You typically use the changing value of the loop variable inside this code so that the loop does something different every iteration, such as looking in every successive index of an array.

For an example of a `for` loop, I'll calculate the factorial of four. The factorial of *x* is obtained by multiplying every integer between 1 and the argument *x*, so the factorial of four is 4*3*2*1=24.

```
var x:int = 4;
trace("the factorial of", x, "is:");
for (var factorial:int = 1; x > 0; x--) {
  factorial *= x;
}
trace(factorial); //24
```

In this example, you multiply every number between 1 and 4 using a loop to count between these numbers. In the loop, `x` decreases in increments of one (`x--`) as long as it's positive (`x > 0`). Every time, the current value of `x` is multiplied with `factorial`, a running total that starts at 1.

Using for..in and for each..in

`for` loops can be useful for repeating an action multiple times, especially when the number of repetitions is known. However, once you master data structures such as `Array`, `Object`, and `XMLList`, you'll find yourself using loops to iterate through collections of data all the time. Depending on the circumstance, this can be greatly aided by several `for` loop variants in ActionScript 3.0. I'll introduce these loops here, but you'll return to them when you learn about `Array` and `Object`.

for..in

The `for..in` loop uses the properties of an `Object` to iterate. So, for every object stored within another object, the loop body executes once. This loop will be revisited in Chapter 10, "Objects and Dictionaries," after an in-depth look into `Objects`.

A typical `for..in` loop is depicted here:

```
for (var propertyName:String in targetObject) {
  //do some action
}
```

In this case, you could read this as, "For every property of the object called `targetObject`, do some action," or "Act upon every property of the target object." `targetObject` is the object whose properties you're looping through. The `propertyName` variable is the loop variable, which stores the name of the current property contained in `targetObject` as the loop proceeds through them.

for each..in

There is another variation of the `for` loop in ActionScript 3.0 that is perfect for iterating through certain data types that support it, notably `Array`, `Vector`, `Object`, and `XMLList`. This is the `for each..in` loop, depicted here:

```
for each (var element:Object in targetObject) {
  // do some action
}
```

This is slightly different from the standard `for..in` loop because with `for each..in`, you deal in the data itself rather than an index or a property name that can be used to look up the data in the data structure. This looping structure eliminates the need for code inside the loop body to look up the associated data in the structure, making things just that much more concise. The `element` *is* the property rather than just the name of the property. You'll see this loop structure again in Chapter 8, "Arrays."

Using while and do..while

The final type of loop that I'll look at is the `while` loop. `while` is a much more basic type of loop than `for`. You can use `while` loops when you need to say, "As long as a certain condition is met, keep looping."

while

This is the basic structure of a `while` loop:

```
while (condition) {
    // do some action
}
```

That's it. Inside the parentheses, put whatever conditional statement you want to test for. As long as the condition is true, the loop body will execute again.

The following two loops are virtually identical:

```
for (var i:int = 0; i < 100; i++) {
    // Do some action
}

var i:int = 0;
while (i < 100) {
    // do some action
    i++;
}
```

`while` loops can be useful when you need to loop without knowing ahead of time when you'll need to stop. For example, this loop adds random numbers together until the result surpasses 100:

```
var sum:Number = 0;
while (sum < 100) {
    trace(sum + " - not there yet.");
    var random:Number = Math.ceil(Math.random() * 10);
    //create a random number between 0 and 10
    sum += random;
}
trace(sum + " - surpassed 100. END");
```

When I ran this, I got the following output. Yours will be different because random numbers are being used.

```
0 - not there yet.
3 - not there yet.
12 - not there yet.
21 - not there yet.
27 - not there yet.
32 - not there yet.
39 - not there yet.
49 - not there yet.
55 - not there yet.
64 - not there yet.
70 - not there yet.
80 - not there yet.
90 - not there yet.
92 - not there yet.
95 - not there yet.
104 - surpassed 100. END
```

Inside a `while` loop is another perfect place to utilize a `break` statement. Recall from the `case` statements that `break` will exit out of a block of code. It can be used to prematurely (or maturely?) terminate a loop. If you really have no way to write a succinct expression that determines the terminal conditions for a loop, you can have the loop continue indefinitely, but interrupted at the proper moment by a well-timed `break`.

```
while (true) {
  //do some stuff
  var userInput:String = getUserInput();
  if (userInput == "exit") break;
  //do some more stuff with the user input
}
```

The preceding code will execute forever as far as the `while` loop is concerned, because I've used `true` as the loop condition, which will always be true, clearly. It's only when the user input matches "exit" that the loop terminates, and it terminates in place, before any of the rest of the loop body is executed. Be careful with this technique. You might want to put in a failsafe that only lets the loop execute a certain maximum number of times, just to be sure. You'll see the scary possibilities of an infinite loop next.

do..while

The standard `while` loop comes in one additional exciting flavor — the `do..while` loop. The only difference here is that the check for the condition happens after the loop is complete instead of before. Consequentially, the loop body always runs at least once — just like the postman. Or something.

```
var sum:Number = 10;
do {
  sum += 40;
} while (sum < 5);
trace(sum + " - surpassed 5"); // Displays: 50 - surpassed 5
```

Even though the loop condition for the previous loop starts out false, you'll never know. It's not even tested until `sum` is way up to 50.

Battle of the Loop Structures: for vs. while

Most developers choose `for` loops over `while` loops for most situations. You should certainly know about both and choose the one that makes the most sense for each situation. If you're not sure, just stick with `for` loops. Either can be used for most situations. `for` statements express a lot more information about the loop in a single line, and are less likely to result in an infinite loop because they tend to count toward a well-defined upper (or lower) limit. Furthermore, although I've shown a basic `for` loop that increments a single loop variable up to its limit, lots of interesting variations are possible. The three loop control expressions can really be anything. If you want to get tricky, you can use a comma operator in the loop initialization expression or the loop counter expression to initialize and operate on multiple loop variables at once.

Avoiding Infinite Loops

If any loop is allowed to continue running unchecked, you are likely to encounter an infinite loop. An infinite loop can occur when the loop variables fail to change or when the loop termination condition is poorly designed so that it never occurs. Infinite loops should always be avoided as they cause the Flash Player to hang. The following is a simple example of an infinite loop.

Caution

Only run this code if you're ready and willing to forcefully terminate Flash Player.

```
while (true) {
   trace("desu");
}
```

Infinite loops don't have to be as obvious as `while(true)` either. Any loop structure can be infinite if there's a mistake in its logic, for example:

```
for (var i:Number = 100; i > 0; i++) {
  trace("desu");
}
```

Using break and continue

In some situations you may find that your loop is no longer useful or you've found the result you were looking for. In these cases you can use `continue` and `break`, respectively.

continue

`continue` ceases to execute the current iteration of the loop body and skips ahead to the next iteration. Use it when the loop body is long or intensive and you can rule out early the need to perform the remainder of the loop body.

break

`break` stops the current iteration of the loop body and terminates the loop outright. Use it when the loop has fulfilled its purpose before the loop condition fails. `break` is especially useful in loops that search for something.

Say you want to search through a bunch of `haystacks` to find `"needle"`. Since I haven't introduced data structures yet, I'll use some made-up methods to look for the needle. Inside every loop iteration, if you dig up only `"hay"`, you can use `continue` to skip to the next iteration until you've identified the needle, and then `break` the loop so that unnecessary repetitions are halted.

```
var stuff:String, needle:String;
for (var i:int = 0; i < numberOfHaystacks; i++) {
  stuff = lookInHaystack(i);
  if (stuff == "needle") {
    needle = stuff;
    break;
  }
}
if (needle) {
  trace("I found the needle!");
} else {
  trace("I couldn't find the needle.");
}
```

Summary

- ActionScript code is plaintext contained in an ActionScript file, usually a class file.
- ActionScript code consists of statements and expressions gathered into blocks of code.
- Whitespace and comments are ignored by the compiler, so you can use them to improve the readability of your code and ease its comprehension.
- ActionScript gets its power from the use of variables, which are essentially containers for holding data for the duration of your program.
- Each variable has a type that defines the data that can be stored inside it.
- Operators are atomic functions used for common applications such as doing basic math and comparing values.
- You can add decision-making logic to your program through the use of `if` statements and other conditionals.
- Repeating the same code multiple times is simplified by using `for` and `while` loops and their variants.

Functions and Methods

Now that you know all about creating variables, you probably want to start actually *doing* something with them. This is where functions and methods come in. *Functions* are reusable blocks of code, also called *methods* when they are part of a class. Functions allow you to organize your code into independent pieces of functionality. They can be used to return various results based on input you provide. Perhaps most important, functions can encapsulate behaviors within a class and provide a public interface to those behaviors — an organized way for other code to access those behaviors. This chapter covers ways to use functions and create your own from scratch.

If there's one thing to remember about ActionScript, it's that every variable and part of a class is an object. Functions are no exception. While it might be strange to imagine, functions are instances of the `Function` class and contain their own methods and properties. In the section "Functions as Objects," you'll learn more about this.

Calling Functions

At the most basic level, a function is a well-defined piece of code, a sequence of statements in a block that has a beginning and an end. A function wraps up these sequences, and you run them at will. Every program you'll write will depend heavily on functions.

Executing the code within a function is known as *calling* or *invoking* the function. Functions in ActionScript are called by using the function's name followed by a pair of parentheses ( ). Officially, these parentheses are known as the *call operator*.

Additional information can be passed on to your functions by adding *arguments* or *parameters* inside the parentheses and separating them by commas. Some functions have optional or required arguments. The number and type of values passed in must match the function definition, also called a *method signature*. Other functions require no arguments and are called by using an empty pair of parentheses.

These are both valid function calls:

```
trace("Hello world!");
addChild(mySprite);
```

Tip

Be careful not to forget the parentheses operator when calling a function. Doing so is not an error, and the compiler won't always catch it! An expression with the function name and no call operator will evaluate the function as a variable (an object of type `Function`) rather than executing the code within the function, producing unexpected results. You will see later in the chapter how and when to use functions without the parentheses operator. ■

You'll learn more about how methods and functions differ in Chapter 4, "Object Oriented Programming." I'll just say here that you'll hear the terms *method* and *function* tossed about without much care, but they are distinct. A method is a function that's part of a class.

Note

Some ActionScript operators, such as `new` and `delete`, are not technically functions even though they appear to behave the same way. As such, you do not need to add parentheses around the arguments for these commands and won't be able to access `Function` methods and properties for these reserved words. Consequently, these operators, unlike most functions, exist globally and will be available anywhere in your code. ■

Creating Custom Functions

To create your own functions, you add function declarations to your ActionScript source code. This is called *defining* or *declaring* a function. Let's look at how this is done.

Defining a Function

Function declarations share this basic structure:

```
function doSomething(arg1:TypeA, arg2:TypeB):ReturnType {
  //the executed code goes here.
}
```

Let's break it down one word at a time.

- `function` — First is the `function` keyword. This is always required when defining a function, just as `var` is always required when defining a variable.

- `doSomething` — Immediately after the `function` keyword is the name of your function. This is the command you will call when you want to run the function.

Tip

The best function names are descriptive, describe an action, and can be read easily as though you were reading a sentence. A good rule of thumb is that your function name will start with a verb. For example, `button()` is not as useful as `drawButton()`. Likewise, the best variable names are usually nouns. Together, you can combine verb and noun to create a short sentence such as `snakeHandler.feed(python, mouse)`. ■

- `(arg1:TypeA, arg2:TypeB)` — Following the function name is the comma-separated list of arguments inside a pair of parentheses. Each argument should be followed by a colon and the data type for the argument. You'll investigate the argument list in more detail shortly.

- `:ReturnType` — After the parentheses are another colon and the data type of the value the function returns when it is done executing.

- `{...}` — The block of code that's executed when the function is called is contained within the two curly braces `{}`. This is known as the *function body*.

All functions require the `function` keyword, the function name, the parentheses, and the function body. The rest of the parts are not required, but that doesn't mean you shouldn't use them.

Passing Arguments to Your Function

Functions become much more interesting and much more useful when you provide them with some external input. Functions are like machines that follow a well-defined set of steps. A machine is far more useful when you can give it different kinds of raw materials to operate with. This can be achieved by adding *arguments*, also known as *parameters*, to your function definition. To do this, simply list one or more arguments within the parentheses of a function statement. The names you define here will be available in the function body as locally scoped variables that you can use to execute your code.

Not all functions will require parameters. Those functions will be invoked with nothing between the parentheses.

This function calculates the circumference of a circle:

```
function getCircumference(diameter:Number):Number {
  return Math.PI * diameter;
}
```

The parameter names — `diameter` in this case — exist in the local scope for the function. Here you see that the variable `diameter` is used in the line inside the function body without being declared with `var`. The argument list serves to declare these variables, their names, and types.

Passing by Reference or by Value

ActionScript 3.0 handles primitive data types and complex data types differently when it comes to passing values into a function. Primitive data types are passed *by value* — that is, their value is copied into the function, leaving the original value intact and unchanged despite what may happen within the function. Complex data types pass values *by reference*, which uses the actual object passed in instead of a duplicate. Incidentally, these rules apply to variable assignments, too.

If you use a computer, you're likely to be aware of the difference between copying files and creating shortcuts (or aliases). As shown in Figure 3-1, passing by value is a lot like duplicating a file because the original file remains where it is, unchanged. Passing by reference is more like creating a shortcut to the original value. If you create a shortcut to a text file, open the shortcut, and edit it, your changes are saved in the original file that you linked to.

The following is an example of passing by value. Notice that the original value doesn't change:

```
function limitPercentage(percentage:Number):Number {
  //ensure the percentage is less than 100
  percentage = Math.min(percentage, 100);
```
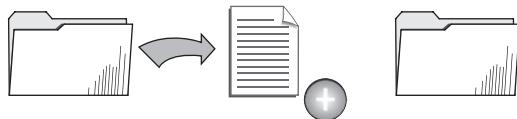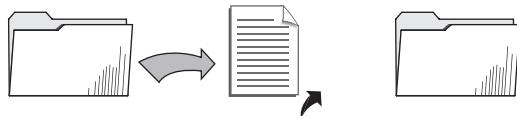
```
  //ensure the percentage is greater than 0
  percentage = Math.max(percentage, 0);
  return percentage;
}

var effort:Number = 110;
var trimmedPercentage:Number = limitPercentage(effort);

trace(effort, "%"); // Displays: 110%
trace(trimmedPercentage, "%"); // Displays: 100%
```

FIGURE 3-1

Copying a file is like passing a parameter by *value*, whereas creating a shortcut is like passing a parameter by *reference*.

Passing by Value

Passing by Reference

The original value for `effort` hasn't changed even though the `percentage` argument variable was reassigned during the course of the function.

The opposite is true for complex values. They are passed in *by reference*, meaning that the argument acts like a shortcut or link to the original value. Changes to the argument are reflected as changes directly on the value that was passed in. Most data types pass references to variable assignments. The following shows how `employee` is directly linked to the object passed in as a parameter.

```
function capitalizeEmployeeName(employee:Object):void {
  if (employee.name != null) {
    employee.name = employee.name.toUpperCase();
  }
}

var person:Object = {name:"Peter Gibbons"};
capitalizeEmployeeName(person);
trace(person.name); //PETER GIBBONS
```

As you can see in this example, the name capitalization is happening directly on `employee`. That's because the `employee` value refers to the original copy of `person` — hence, the person's name is linked as well.

The following "primitive" data types are passed by value:

```
String
Number
int
uint
Boolean
```

All other data types are passed by reference.

Setting Default Values

You can set default values for a method's arguments. Adding a default value for a parameter allows you to omit the parameter when calling the function — and when omitted, the parameter takes on the default value. To do this, simply add an equal sign (=) and the default value after an argument name, as follows:

```
function showGreeting(name:String = "stranger"):void {
  trace("Hello," + name + ", nice to meet you.");
}
showGreeting("Mr. Whimsy"); //Hello, Mr. Whimsy, nice to meet you.
showGreeting(); //Hello, stranger, nice to meet you.
```

As you can see, in the second call, the `name` parameter is omitted when calling the function, and the function uses the default value of `"stranger"`.

There is one other rule to keep in mind when using default values. Because all values with a default are optional, they must be placed last in the order of arguments, and you can't follow an omitted argument with an included argument. Because arguments are positional, it's not clear when or where one's been omitted unless you omit every argument after a certain point. The following code

```
function storeAddress(name:String, zip:String = null, email:String)
```

is not valid because `email` is a required parameter and it appears after `zip`, which is optional. The correct order would be

```
function storeAddress(name:String, email:String, zip:String = null)
```

Using the Rest Parameter (…)

The *rest parameter* (`...`) is a keyword that accepts a variable number of additional arguments. You follow this keyword with an argument list name, which will be used inside the function body to access the additional arguments — they will be stored as an `Array`. Adding a rest parameter to your function allows you to make calls with as many arguments as you like. For example, if you want a function that adds any quantity of numbers, you might use the rest parameter:

```
function sum(... numbers):Number {
  var result:Number = 0;
  for each (var num:Number in numbers) {
    result += num;
  }
  return result;
}

trace(sum(1,2,3,4,5)); //15
```

43

The values passed in to the function are contained within the argument list called `numbers`. When you loop through the array, you can access each value.

Cross-Reference

To make the rest parameter useful, you'll need to understand Arrays. You'll find all you need in Chapter 8, "Arrays." ■

The rest parameter can also be used with other required arguments. The required parameters will have their own names as usual and will exist independently of the variable argument list. Any additional parameters after the required ones will be stored in the argument list. Let's modify the previous example so that it requires at least one argument:

```
function sum(base:Number, ... numbers):Number {
  var result:Number = base;
  for each (var num:Number in numbers) {
    result += num;
  }
  return result;
}

trace(sum()); //Error!
trace(sum(1,2,3,4,5)); //15
```

Returning Results
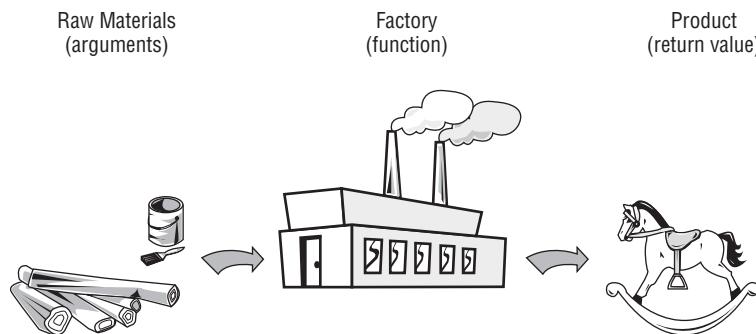
Functions can do much more than simply run predefined code — that's only half of their purpose. Like mathematical functions, functions in ActionScript can calculate values based on variable input. This ability is extremely useful.

You can think of a function as being a factory that can convert raw materials into a finished product, as illustrated in Figure 3-2. In this case, the raw materials are the arguments that you pass into a function call, and the finished product is the value that the function returns.

FIGURE 3-2

Functions, like factories, convert raw materials into a finished product.

Raw Materials   Factory   Product
(arguments)   (function)   (return value)

Returning a Value Using a return Statement

In its simplest form, a *return statement* is the keyword `return` followed by the value that you want to return. This value can be a literal, a variable, the result of another function, or any other type of expression.

```
return someValue;
```

Notice that no parentheses are used in conjunction with the `return` statement. This is because it is an operator, not a function. You may see parentheses used sometimes, as in `return (someValue)`, but they are not required. In this case the purpose of the parentheses is not to invoke a function but to group an expression, as in `(1+2)*3`.

In most situations, you'll want to manipulate the arguments within your function somehow and then return a new calculated value. This example calculates the circumference of a circle by taking a single variable, `diameter`, multiplying it by the mathematical constant π, and returning the result as a number:

```
function getCircumference (diameter:Number):Number {
  var circumference:Number = diameter * Math.PI;
  return circumference;
}
trace(getCircumference(25)); //78.53981633974483
```

As you can see, the `getCircumference()` method call can be used in a `trace` statement as if it were a variable. Results from a function with a return type can also be saved to a variable by putting the variable on the left side of the equal sign.

Whenever a `return` statement is executed in a function, the function ends and the rest of the code is not processed. Because of this, only a single value may be returned for each function. A `return` statement can also be used to end a function prematurely.

```
function getColor():String {
  var color:String = "Red";
  return color;
  color = "Blue";
  return color;
}

var color:String = getColor();
trace(color); // Displays: Red
```

This method always returns `"Red"` because the function code stops executing abruptly after the first `return` statement.

Tip

To return several values from a function, you may want to create an object or a composite data type to hold all your answers in a single variable. You'll learn about how `Object` **can assist with functions in Chapter 10, "Objects and Dictionaries."**

```
return {color:"Blue", arms:8};
```

Defining a Return Type for Your Function

When you define a data type for a variable, constant, or argument with the colon (:) after the object name, it indicates to the compiler what kind of information should be stored in the object. Functions are the same way.

All functions that define a *return type* must honor it by returning a value with a `return` statement that matches the data type. Likewise, all functions that use a `return` statement should have a return type. A return type, like the type attached to each argument, tells the compiler what kind of data to expect when the function returns a value.

Tip

Constructors are a special case. You might be tempted to use the class name as the return type for your constructor, but ActionScript dictates that you should leave the return type for all constructors blank — for example, `public function MyClass() {...}`. For more information on constructors, refer to Chapter 4. ■

Returning void

If you don't need your function to return anything, simply use a return type of `void`. No `return` statement is necessary for functions that return void.

```
function doNothing():void {
  //Okay, I won't do anything.
}
```

Even though this example "does nothing," functions that don't return values are common. Just because a function doesn't return a value doesn't mean it can't execute useful code. In fact, a great number of functions in your code are likely to have no return statement. These are sometimes called *procedures* or *subroutines* — small, reusable blocks of code that serve to encapsulate frequently used tasks.

Anonymous Functions

So far, you've looked at defining functions using a `function` statement. There is an alternative way to define functions; ActionScript also supports *anonymous functions* or *function expressions*. Anonymous functions are so called because they have no name. They can be passed around, stored in variables, and run by other code. Anonymous functions are particularly useful when using a programming style called *functional programming*; you won't see much of them until Chapter 8, "Arrays," and only infrequently thereafter.

To create and store an anonymous function, write a function statement, leaving out the name, and then assign it to a variable of your choosing with a type of `Function`, as shown:

```
var doSomething:Function = function (arg:Object):void {
  // function code goes here.
}
```

Anonymous functions behave slightly differently from function declarations. Most importantly, they are an expression rather than a declaration. They define — and evaluate to — an object of type `Function`.

Table 3-1 compares some of the key differences.

TABLE 3-1

Differences between Declared and Anonymous Functions

Declared function	Anonymous function
Can be called at any point in the code regardless of where in the code the function is defined.	Like a variable declaration, is available only in the scope the function is defined.
Can be called as a method by using dot syntax; acts as part of the object's interface.	Cannot be used as a method.
Belongs to the object it's defined on.	Exists in scope it's defined in.
Cannot be rewritten or deleted.	Anonymous functions stored in variables can be reassigned a new value or deleted.

For most situations, I recommend that you use function declarations rather than anonymous functions in your code. Their consistency and ease of use make them the superior choice. However, sometimes using a function expression can provide flexibility and usefulness that a function statement cannot. These may include the following:

- Functions that need to be used only once
- Functions useful only within another function
- Changing the functionality of a method at runtime
- Passing a function
- Using functional programming paradigms like `Array.map()`

Functions as Objects

Almost everything in ActionScript is an object. Functions are no exception. Every function in Action-Script 3.0 is an instance of `Function`, with properties and methods just like any other object. In this section you'll look at how this relationship can be exploited for your benefit.

The word "function," written in lowercase, is the generic term that I've been using so far in this chapter to describe an executable block of code. `Function`, written capitalized, is the `Function` class that all functions are instances of. Normally when you create a new instance of a class you use the `new` keyword followed by the class name. However, because functions are such an integral and frequently used part of ActionScript, there are commands (like the `function` keyword) that are used instead of the `new Function()` statement.

Note
Technically, you can use `new Function()`, but it doesn't do much except create an undefined function. ∎

You are most likely to encounter the `Function` class by name when working with methods that require functions as input or when creating functions by expression. For example, the method `addEventListener()` takes a function as a parameter. If you're not familiar with events, don't worry about it. I'm going to cover them in great detail in Part IV of this book, "Event-Driven Programming." For now, let's look at what arguments it's expecting.

```
addEventListener(type:String, listener:Function):void
```

All told, `addEventListener()` accepts five arguments. I've included only the two required ones for the sake of simplicity.

As you can see, the second parameter expected by `addEventListener()` is a listener function. This is the function that executes when the event is detected. If you want to pass a function into another function as an argument, simply use the passed-in function name but leave off the parentheses (call operator).

```
function onClick(event:MouseEvent):void {
  //Handle click.
}
addEventListener(MouseEvent.CLICK, onClick);
```

Tip
Be careful here to omit the parentheses. If you had written

```
addEventListener(MouseEvent.CLICK, onClick());
```

the `onClick()` would have executed and returned `void`, which would be passed as your parameter (and `void`, of course, is not a function). ■

Recursive Functions

By now I've covered all the essential parts of a function. But I haven't talked much about how to craft a function. Part of this is because there's really no single formula. Each function should try to do just one thing, and of course its name should reflect that thing. Your functions should do whatever is necessary to carry out their mandate, taking whatever arguments they need to do their job and returning a value when necessary.

However, one particular form of function bears mentioning here. Sometimes a function needs to be self-referential, in which the result of the function actually depends on the application of the same function to a smaller part of the input. This somewhat tricky form of function design is known as *recursion*.

Here are some instances in which you would use a recursive function:

- When stepping through deep data structures like trees or connected graphs
- When calculating mathematical functions that are fractal or recursive in their own structure

- Functions that require back-stepping to previous iterations or that step through a complicated set of possible outcomes, such as a maze-solving algorithm
- When you can solve a problem by reducing the data down to a trivial state and then applying the results back up to the original data

Let's take a look at a sample recursive function — the factorial. The factorial, usually written as a positive integer followed by an exclamation point (4!), is a function that calculates the product of all the positive integers less than or equal to an integer. In other words, 3! = 3 * 2 * 1 = 6. You could write this using a recursive function, as follows:

```
function factorial(i:int):int {
  if (i == 0) {
    return 1;
  } else {
    return (i * factorial(i - 1));
  }
}

trace(factorial(3)); //6
```

What happens when you run this function? The original value 3 is passed in. The value 3 is greater than 0, so you skip over the first condition to the `else` statement. This returns `i` times the factorial of `i-1` (2). But before returning the value, the factorial statement kicks off for a second time, using 2 as the parameter. Then the whole process repeats. `factorial(2)` returns `i * factorial(i-1)` (or 2 * the factorial of 1). This repeats until all the integers are processed and the recursion ends, finally sending all the returned values to be multiplied together by the function.

When creating a recursive function like this, be sure to ask yourself: can I do the same thing with a loop? If you can replace recursion with looping without adding too much to the algorithm, you should do so. Looping is generally more efficient and easier to debug than recursion. In the factorial example, you could use a variable to accumulate the running product, eliminating the need for a recursive call. Take a look at how the recursive call operates for a hint. If there is only one recursive call, you should be able to easily convert the function to a loop structure.

```
function factorial(i:int):int {
  for (var product:int = 1; i > 0; i--) {
    product *= i;
  }
  return product;
}

trace(factorial(3)); //6
```

While working with recursive functions, you might encounter a *stack overflow* issue. This problem occurs when a recursive function has no mechanism that stops it from calling itself forever. Flash Player has a built-in mechanism that limits how many active function calls it can remember. If your recursive function needs to recurse too many times, you may get an error such as this one.

```
Error: Error #1502: A script has executed for longer than the default
timeout period of 15 seconds.
```

Summary

- Methods and functions are repeatable actions that execute blocks of code based on the input parameters that are passed.
- Using the `return` operator, you can calculate and return values based on variable input with your functions.
- Like most other things in ActionScript 3.0, methods are objects. They belong to the `Function` class.
- Functions can be constructed using `function` statements or by assigning a value to an object of type `Function` using function expressions.
- The `...` (rest) parameter can be used to access all the arguments of a function if the number of arguments is unknown or variable.
- A subclass can use the `super` and `override` keywords to add on to the functionality of a superclass's methods.
- Recursive functions allow you to loop an action and build on the results that were generated from the previous call.

Object Oriented Programming

Programming using classes is essential to take advantage of the full power of ActionScript 3.0. By using classes, you participate in *object oriented programming*, a versatile way to think about and structure code. Objects are high-level building blocks, and classes are the constructs that define them. In this chapter you learn what constitutes a class, how classes relate to objects and other language constructs, and how to write a class in ActionScript 3.0.

Understanding Classes

To program using classes, you must understand what classes are. Classes perform multiple roles in object oriented programming, and there are many ways to think about them. This section introduces these fundamental ideas.

Classes Can Model the Real World

Classes often work best and are easiest to think about when they represent objects in the real world. You can, for example, create a class to represent a bicycle. That bicycle can have a color, a size, and two tires. You can pedal it, brake it, switch gears, and ring the bell. The bicycle has relationships with other real-world objects: your feet, the road surface, and occasionally a bike pump. If you model the bicycle as a class, you can represent all of these and carefully design it to interact with the other objects in the right ways. Let's call this class `Bicycle`. I will come back to the `Bicycle` class to describe concepts in class design throughout this chapter.

You can use classes throughout your program to model tangible objects within the world of your program, if you are modeling actual phenomena. However, classes do equally well representing abstract concepts and nonphysical objects. You might consider, for example, a table in a database as a class. Although nobody has actually seen a database table with their eyes, or knows what it sounds like or tastes like, you can form a complete representation of it.

51

You know everything that a table must do in its own world and all the properties that define it. Therefore, you can easily model it as a class.

When you become comfortable associating real-world objects with classes in your program, you will have the basics of object oriented programming under your belt. You'll be modeling `Toast`, `Guitar`, `LaserGrenade`, and `FuzzyLobster` classes with aplomb.

Classes Contain Data and Operations

The textbook definition of a class is a structure that contains both data and operations. Classical organization in object oriented programming differs from *procedural programming*, in which data and operations (variables and functions) mingle without supervision. Imagine placing the color of a bicycle next to the action of popping a water balloon next to the number of licks in a lollipop. You could make sense of procedural programs only if you were strict about imposing your own organization. Classes group the activities and properties associated with the same subject together, and impose order naturally. The color and speed of a bicycle, the actions of braking and pedaling appear together in a `Bicycle` class, and the flavor, color, and duration of a lollipop appear together in a `Lollipop` class.

Note
Frame scripts and ActionScript 1.0 are examples of procedural programming. In procedural programming, code executes from top to bottom and can go into functions or procedures. ∎

Binding the behaviors of an object with its properties results in a self-contained, fully functional, organized unit. In the same breath, you can talk about rotating the pedals of a bicycle and the color of that bicycle. It's all the same bicycle.

Classes Separate Responsibilities

Classes can perform work. You use them when designing a program to determine how to split up the program's responsibilities. Say you are required to build a program that makes deliveries throughout the city. You have to handle the responsibilities of picking up an item, finding out its destination, determining the best route to that destination, handing off the package, and collecting a fee. The `Bicycle` class, then, is just one class in your program. When thinking about how to do a program with object oriented design, you create classes to perform certain responsibilities. Here, a `Dispatcher` would be a natural choice for determining optimal routes, and a `Messenger` would be needed to control the `Bicycle` and to handle the `Package`. By carefully analyzing the actions that your program needs to perform, you can determine what classes are appropriate to use. With just a little more thought, the relationships between these classes become obvious as well.

Classes work well when they are small, specific units. If you end up with a class that takes care of many different responsibilities, it's usually possible and desirable to split it into multiple classes. If you write a `Level` class that draws a centipede, player, and obstacles and keeps track of the score, you've created the same kind of jumble as exists in a procedural program. Look instead at the nouns used in that description for your clues to what the classes could be: a `Centipede`, a `Player`, a `Stage`, and a `Score`.

Classes Are Types

ActionScript 3.0 uses classes as types. All `Bicycle` classes are of a type `Bicycle`. This isn't because of any confusing self-reference but simply the nature of *names*. Bicycle is both a definition of a thing (has two wheels, gears, and so on) and the word you use to refer to that thing ("bicycle"). When you

write out a class, you declare its name, and then you define it. When you use that class in the rest of your program, you only have to refer to it by name.

In ActionScript 3.0, the inverse is also true: all types are classes (okay, except for interfaces, which you'll learn later in this chapter). In some languages, simple types like numbers are not represented by classes. These are called *base types*. In a language like this, 12 would have a type, say, `integer`, but no class associated with it. The type of 12 has a name but no definition. There's nothing more you can learn about 12; there's no data contained in it or operations it can perform. However, in Action-Script 3.0, even numbers like 42 and Boolean values like `true` are represented by classes. They have data — at minimum, their own value — and operations. For example, you can ask a number like 12 to represent itself as a string by calling its `toString()` operation. In ActionScript 3.0, *every* value is described by a class.

Classes Contain Your Program

With the new generation of tools that support ActionScript 3.0, you are encouraged to write all your code in classes. There is no functionality that can't be performed with the right objects acting together, and creating a set of classes that divide up all the responsibilities of your program is what object oriented design is all about.

This means that scripts attached to frames or symbols using Flash Professional can be, and should be, written in appropriate classes rather than on the timeline. Even the initialization of your program can move out of frame scripts by associating a Document class to your Flash project. All runnable example code in this book is contained in a class, even if that class simply runs a few lines of ActionScript.

In ActionScript, classes are stored in `.as` files. Most of the time, each class file contains a single class (although this is not a requirement, as you will see later). ActionScript class files end in the file extension `.as`; for example, `Bicycle.as` might contain the code for the `Bicycle` class.

Note
The `.as` extension is used for ActionScript 1.0, 2.0, and 3.0. Accordingly, simply examining the filename will not tell you what language is contained in the file. ActionScript 3.0 files can be distinguished by the `package` block that surrounds the class definition. ∎

Object Oriented Terminology

Although classes are the cornerstone of object oriented programming, there are other essential, related concepts. I've already touched on some, and I've tiptoed around others, for it's impossible to discuss object oriented programming without this vocabulary.

Object

Object can have slightly different meanings in different contexts. An object is the basic unit of object oriented programming. It is a self-contained thing that contains data and operations. Every object is defined by a class. For example, a bicycle object would be defined by the class `Bicycle`.

An object is concrete. A real bicycle has two wheels that make it a bicycle — a bicycle object has traits that make it a `Bicycle`. Any instance of `Bicycle` is a bicycle.

Typically, you use the word *object* to refer to an arbitrary object — one whose details you are not interested in. When the object has a specific type, sometimes you use the word *instance* instead, as in "this is an instance of the `Bicycle` class."

Objects are constructs that exist at runtime, while the program is running. You can write code to manipulate objects, but the objects don't actually exist until the code you wrote executes and creates them.

The word *object* can also refer to the class `Object`, which is the base class of all classes and the root of all type hierarchies. You can find more about this later in this chapter and in Chapter 10, "Objects and Dictionaries."

Class

A *class* is a blueprint for an object. It defines the object in full: the data and operations of that object. You write classes before you run your program, and when your program is running, the classes are set in stone. An exception to this is dynamic classes, which are discussed later in this chapter. Otherwise, the only thing that a running program uses classes for is to create new objects and to manipulate types.

Classes contain the code you write. When your program runs, it's an orchestrated production of objects, and that production is directed by the code in the classes.

Instance

An *instance* is a specific object created from a specific class. It may be used by itself or with the name of the class to specify the type of the instance, as in "an instance of the `FuzzyLobster` class." Instances are unique. When you point to a particular shiny red road bike screaming down the road, you can refer to an instance of `Bicycle`. Although there are other bicycles, and although there are other shiny red road bikes, that instance is one individual bike.

The word *instance* is related to the word *instantiate*. When you create a new instance of a `Bicycle`, perhaps with the code `new Bicycle()`, you are also instantiating a class.

Often, instance and object are used interchangeably. All instances are objects. Usually, you deal with objects (like a particular bicycle) that are instances of a certain class. In most cases, the terms *object* and *instance* are equally valid.

Type

The *type* of an object tells you what the object is. An object of type `Bicycle` is a bicycle. ActionScript 3.0 allows you to create variables with and without types. In other words, you can create both *statically typed* and *dynamically typed* variables. Except where the situation requires it, all code in this book will use statically typed variables.

In statically typed code, types help to ensure that you only perform actions on the right kind of things. For example, if you want to do a wheelie on a bicycle, you might make sure that something is a bicycle before you try to pop a wheelie with it. Imagine the embarrassment you would have to endure if you were caught popping a wheelie on a strawberry, or the hospital bill you'd incur by popping a wheelie on a unicycle. When you write statically typed programs, the code can enforce these checks; your code will be incorrect if it uses incompatible types, and the compiler won't let you run the program. The assurance that types are correct is known as *type safety*.

Because ActionScript is object oriented, the actual type of every object will be a class, and it is impossible to create an object with a type whose type is not a class. (You can use objects as if they were compatible types and interfaces, as you will read later in the section "Manipulating Types," but their actual type will remain the class they were instantiated from.) From `Strings` to `ints`, objects' types are classes.

Encapsulation: Classes Are Like, So Selfish

By containing and controlling access to the data that represents it, an object can hide its implementation from the outside world. Outside that class, no other part of the program can see how it works internally. This principle is known as *encapsulation*.

Encapsulation is also described as *information hiding*. Classes should keep certain information pertaining to their implementation hidden from the outside. This practice is at work constantly in the real world. You don't need to know how a bicycle works to ride it, understand the principles of electromagnetic wave propagation to use a microwave, or comprehend how the telephone system is structured to make a phone call. If you had to deal with the information that is hidden by the systems you use every day, you'd probably go completely insane. Encapsulation allows other objects in the program to be users of an object in the simplest way possible.

You also depend on the *interfaces* of systems to be able to use them. While there are some weird bikes out there, let's agree that bicycles have two pedals that you can apply force to, a handlebar to change direction, some manner of controls to change the gears, an indicator that shows what gear you're in, levers to squeeze and apply the brakes, and a lever which rings the bell. This is the interface of the bicycle: everything about it that you can affect and inspect. It is all you need to know about to use a bicycle. Similarly, telephones expose an interface of 12 keys, and microwaves have several functions you can use to cook and reheat your food.
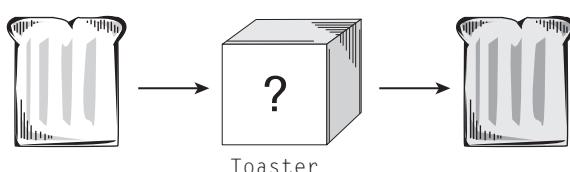
The Black Box Principle

In the real world, you may be able to see the gears of a bicycle, see how the brakes work, and touch and modify parts of the bike you didn't mention in its interface. But in programming, you deal with abstractions of reality. The goal is to hide anything about an object that the user of that object does not have an explicit need to know or use. In the real world, hiding the brakes and the derailleur of a bike might cost more money or make it harder to repair, but when you're designing a program, you use encapsulation with the primary user of the object — in this case, the bike rider — in mind.

Hiding an object's implementation and only providing access to an interface you carefully design is known as the *black box principle*, illustrated in Figure 4-1. If the user wants to make some toast, you can provide her with a sealed black box with inputs and outputs. You put in a slice of bread, and you get toast. You don't need to know or care how the bread is transformed into toast. If some new, more efficient way of toasting arrives, you can get a new black box, and it will still toast your bread; you won't care.

FIGURE 4-1

A toaster. How does it work? Who cares?!

Toaster

Encapsulation and Polymorphism

The prior example touches on another principle of object oriented programming. If your goal is to get toast, you can be equally happy using a `HeatedResistorToaster` object or an `AcetyleneFlameToaster` object, as long as they both have a slot that accepts bread and they produce acceptable toast. Similarly, provided you know how to ride one kind of bike, you can quickly start riding your friend's bike, even if it's a mountain bike and your bike is a road bike. The principle being hinted at by this real-world equivalent is *polymorphism*, or the ability of related classes to be substituted for each other.

More specifically, class instances may always be referred to by a type more general than their actual type. This means that both a `RoadBike` and a `MountainBike` may both be referred to as `Bicycle`, and the parts of their interfaces that belong to the more general `Bicycle` may be used on an instance of either. You can ride both `RoadBikes` and `MountainBikes` if you treat them as `Bicycles` and use the common `Bicycle` interface on them.

The ActionScript 3.0 language gives you tools to create objects that act as black boxes, hide their implementations, and provide sensible interfaces. But it doesn't require that you use the tools correctly: you can violate all these principles while writing legal code. As the programmer, you shoulder the responsibility for upholding these principles in classes you write. Keep the principles of encapsulation in mind, and this chapter will show you how to embody them in code.

Packages: Classes, Functions, and Packing Peanuts

Simply separating your code into classes isn't always enough to keep the code your program requires in order. Large software projects can contain lots of classes — hundreds or even thousands of them — that beg to be organized further into related groups. The functionality of Flash Player is exposed in dozens of classes, which are also grouped by function. For example, `flash.net` contains classes related to network access, and `flash.geom` contains classes that help with geometry.

`flash.net` and `flash.geom` are examples of *packages*. Packages are structures that group classes together, define the full name for the classes they contain, and add a level of access control. Packages can be nested arbitrarily deeply using dot notation, as these examples are.

When you refer to classes by their class name, for example, `Rectangle`, you may not be giving their full name. Because you find the `Rectangle` class in the `flash.geom` package, the class has what is known as a *fully qualified* name: `flash.geom.Rectangle`. In a conversation, once you establish that Roger means Roger Eliot Braunstein, and there's only one Roger in your story, you can simply use the name Roger, unless another character appears with the same first name. The same principle applies to writing code.

Class Uniqueness and Namespaces

Without packages to organize classes, you would find yourself with the problem of uniqueness. Say your next project is a drawing tool written in ActionScript, and one of the shapes you have to allow users to draw is a rectangle. It would only be natural to start writing a `Rectangle` class to handle the display of `Rectangle` objects, but — imagining packages don't exist — there is already a `Rectangle` class in the Flash Player API, and it doesn't do what you need it to.

(The `flash.geom.Rectangle` class stores the position and size of a rectangle. It does not draw anything.) You would be left with no option but to name your class something else, so you might grumble to yourself, name it `DrawingRectangle`, and move on. But after hundreds of such decisions, you might start running out of names.

Indulge one more hypothetical and imagine that, after much hard work on the drawing application, you decide to use a third-party ActionScript library to display dialog boxes. To your dismay, you realize that the authors of the library have their own class called `DrawingRectangle` for drawing buttons on the dialog box. Without packages, you have to find this and all the other name collisions between your code and theirs and resolve them, as well as rewriting any code that references a class whose name you changed.

An entire set of unique names is known as a *namespace*. Packages make it possible to create new namespaces. Packages give your classes rational names and enable you to share code with others without worrying about name collisions. As usual, this language feature doesn't bestow its benefits automatically. You have to be careful to use packages and to name them well. Putting all your classes in a package named `classes`, for example, is not likely to guarantee that they will be unique. (Except for the fact that, hopefully, nobody else will make that mistake!)

The typical technique for guaranteeing a package's uniqueness is to piggyback on the uniqueness of domain names: you are likely to own a domain, or the work you're doing is likely going to end up hosted on one. If you were creating your drawing application for `http://www.example.com`, your `Rectangle` class might reside in the package `com.example.shapes`, so that its fully qualified class name is `com.example.shapes.Rectangle`. Core classes that are built into Flash Player, and the Flex framework classes, are special cases, reserving the package names `flash` and `mx`. For example, the `MovieClip` class in AS3 is located in `flash.display`, so that its fully qualified class name is `flash.display.MovieClip`. This is the exception, not the rule. Even for throw-away code, placing your code in packages starting with your domain name in reverse is a good practice to get into.

Note

If you don't have a domain name and your work won't be hosted on one, you don't have to use the reverse domain name convention. The idea is simply to create something that is unique to you. You could use your full name, a made-up domain name that isn't likely to be used, or anything else likely to be unique. ∎

Hierarchy

Packages serve to organize classes into related groups. In the example class `com.example.shapes.Rectangle`, I have created a package for shape classes. I anticipate having to write other shapes, like `com.example.shapes.Circle`, so this package will keep them grouped together. If I were to support both two- and three-dimensional shapes, I could further group these with an intermediary package, keeping `com.example.shapes.twodimensional.Triangle` and `com.example.shapes.threedimensional.Cone` separate. Exercise the same heuristics you might apply when organizing files in your hard drive into directories to section off your code into cohesive groups.

Because packages serve to both guarantee uniqueness and provide organization, packages for both of these purposes are combined into a single package structure. As in the examples you've read already, place the organizational packages inside the package path you used to guarantee uniqueness. So you would interpret `com.example.shapes.twodimensional.Triangle` as a triangle that's a two-dimensional shape (organization) from the domain example.com (uniqueness).

This hierarchy is visible not just inside your code but on your file system. ActionScript classes must be located in nested folders whose names match the package they are contained in: `com.example.shapes.Rectangle` must be defined in the file `com/example/shapes/Rectangle.as` in your source directory. In Flash Professional, the default source directory is the same directory that your FLA resides in, though you can add other directories to your classpath in File ➪ Publish Settings ➪ Flash ➪ ActionScript 3.0 Settings ➪ Source path. In Flash Builder and Flex Builder, the source directory can be set in your project's Properties ➪ ActionScript Build Path ➪ Main source folder.

Controlling Visibility

Packages in ActionScript 3.0 impact the visibility of the items they contain. The listing that follows might be found in the file `com/example/shapes/Rectangle.as`.

```
package com.example.shapes {
  public class Rectangle {
    // define Rectangle here.
  }
}
class SecretClass {
  // define SecretClass here.
}
```

The first line declares the package `com.example.shapes`. Notice how two classes are declared in this file: one inside the package block that matches the name of the file, and one outside the package block.

In any file, a maximum of one class may be made accessible to the world outside that file. This class must be contained in a `package` block, and it must be declared `public`. It is rare to write code outside a `package` block, but this example illustrates the technique with the `SecretClass`. This technique may be used to simulate the effects of nested class definitions, which are available in Java but not ActionScript. `SecretClass` will only be accessible to other code inside the file `Rectangle.as`. Often, the out-of-package classes are helper classes for the public class in the file that the outside world never needs to know about. Even so, most of the time your class files will have a package block containing a single public class.

Note
Runnable examples from this book place all their classes in a single file and use classes outside the package block. This is for ease of distribution and organization of example files and because the online code compiler requires it. ■

For code inside the `package` block, like `Rectangle`, access modifiers — `public` in this example — can determine where the code is usable from. You'll learn all about access modifiers later in the chapter.

Code Allowed in Packages

By far the most common configuration of an ActionScript 3.0 file is one public class contained in a package block. The class is named the same as the file, and the file's path corresponds to its package. While the single-class-per-file setup accounts for a majority of situations, variables, functions, and namespace declarations can also be contained inside packages.

All the same rules apply for files containing variables or functions in package blocks. There can only be one item per file visible to the outside world, whether that item is a class, variable, function, or namespace. The name of the file must match the name of the externally visible item. For example, you can write a file `com/example/shapes/DEFAULT_SIZE.as`:

```
package com.example.shapes {
  public const DEFAULT_SIZE:int = 256;
}
```

This file declares a package-level constant that you can use in any other code, without tying the constant to a specific class.

An example package-level function might be written in `com/example/shapes/testShapes.as`:

```
package com.example.shape {
  public function testShapes():void {
    trace("test the shapes here...");
  }
}
```

An example of a package-level function found in the Flash Player API is `flash.net`
`.navigateToURL()`.

Using Code from Packages

To use code from a package, you must import the code using an `import` statement. By importing the classes (and variables and functions) that you use from other packages, you tell the compiler to fetch the class definitions, and by virtue of the file path mirroring the package, you tell the compiler where to find those definitions. Without importing the code you need from other packages, the compiler will have no idea what you mean when you refer to `Rectangle`.

When you refer to other classes in the same package, there is no need to import them. If you referred to the `Circle` class inside the `Rectangle` class, provided these were both located in the `com.example.shapes` package, you could do so without importing `Circle`, and vice versa.

`import` statements always appear inside the package block but before the class declaration in Action-Script 3.0 files. The following example is a class that demonstrates your ability to import and use code from the package you just created.

```
package {
  import com.example.shapes.Rectangle;
  import com.example.shapes.DEFAULT_SIZE;
  import com.example.shapes.testShapes;

  public class PackagesTest {
    public function PackagesTest() {
      var r:Rectangle = new Rectangle();
      10 + DEFAULT_SIZE;
      testShapes();
    }
  }
}
```

In the example, you use three `import` statements, line after line, to import the class, constant, and function from the `com.example.shapes` package you have been working with. The `import` statements come inside the `package` block but outside the `class` block. Each `import` gives the fully qualified name of the item to import, whether it is a class, function, or constant.

In this example, there is nothing between the `package` keyword and the open brace, where the package name usually goes. That's because this code defines a class in the *default package*. The fully qualified name of the class in the example is `PackagesTest`. Rather than think of this as no package, think of it as the root, or top-level, package. Adding classes to the default package is not recommended, as they lose the benefits of uniqueness, and the top level is already populated by core types of ActionScript 3.0. Nonetheless, it is important to note that even in files in the default package, the `package` block is required.

One exception exists for the default package rule. In Flash Builder and Flex Builder, the application class must be placed in the default package. The application class is the class that is instantiated when you run an application, starting the application. All runnable example classes in this book are in the default package because they are application classes.

Additionally, you can ask the compiler to import all the classes in a particular package by using the wildcard operator (`*`) trailing the package name:

```
package {
  import com.example.shapes.*;

  public class PackagesTest {
    public function PackagesTest() {
      var r:Rectangle = new Rectangle();
      10 + DEFAULT_SIZE;
      testShapes();
    }
  }
}
```

Note
Importing a class does not automatically include that class in your compiled SWF. You can import a class but never reference it, and it will not be added to your compiled project. For a class to be compiled and included, you must actually use it in executable code. ■

After classes have been imported using their fully qualified class names, they can be referred to by their class names alone. Using an `import` statement is analogous to explaining that when you say Roger in subsequent conversation, you mean Roger Eliot Braunstein. This applies equally to all types of imported code, as the previous example shows. The exception to this rule is when two classes with the same name have been imported from different packages. Their unique packages guarantee you can still use both, but when both are imported, you must always disambiguate references to the classes by using their full name. If you had also used the `Rectangle` class from `flash.geom`, your example would change:

```
package {
  import com.example.shapes.Rectangle;
  import flash.geom.Rectangle;

  public class PackagesTest {
    public function PackagesTest() {
```

```
            var a:com.example.shapes.Rectangle = new com.example.shapes.Rectangle();
            var b:flash.geom.Rectangle = new flash.geom.Rectangle();
          }
       }
    }
```

In this case, you lose the benefit of using just `"Rectangle"` to refer to a class. That reference would always be ambiguous between the two `Rectangle` classes imported. Similarly, if there were two Rogers at a party, you would have to use their full names when talking about them to keep them straight.

Using Inheritance
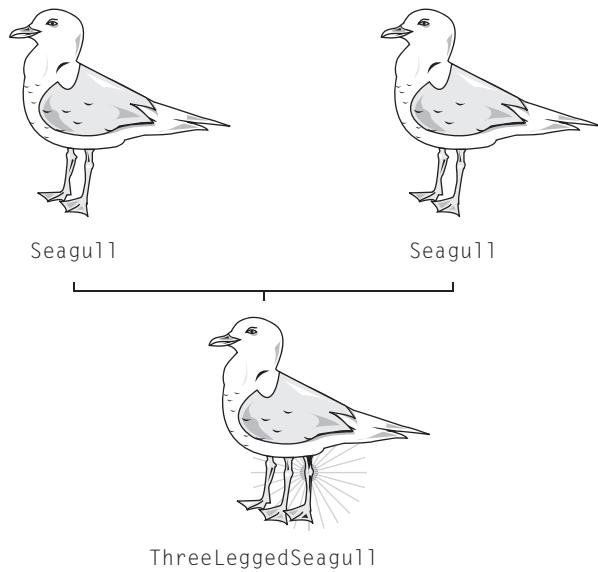
Inheritance is one of the most powerful tools in the object oriented toolbox, and in fact it's a primary motivation for the existence of OOP. It is a core feature of ActionScript 3.0 and other object oriented languages.

Simply put, inheritance is a relationship between classes where one class builds on another. Think, for a moment, of Darwinian evolution. When an organism reproduces, its children inherit the essential attributes of its parents. A seagull will impart to its children two wings, two eyes, feathers, and a beak: all the things that make it a seagull. However, a random mutation or a chance chromosomal configuration can result in the child having a new or modified attribute, like an extra-strong beak or a louder call, or even a third leg, as shown in Figure 4-2. Although the interesting part about Darwin's theory lies in the success of these children, the mechanics of passing on traits suffice to explain inheritance.

FIGURE 4-2

A child inherits traits from its parents, mutations and all.

Seagull   Seagull

ThreeLeggedSeagull

Inheritance in OOP is also a parent-child relationship. A child class, or *subclass*, inherits the attributes of its parent class, or *superclass*. Without any additional work, you can create a class that functions exactly like another class. Because of this, inheritance is a primary way to achieve code reuse. But there's no reason to create an identical copy of a class. Remember, a class is a template, and you can create many instances of it. Inheritance is useful when you use it to *extend* the abilities of an existing class.

Say you have a working `Seagull` class, which is about 2 lbs, able to squawk, fly, and eat sand crabs. For a while it's fun to watch your `Seagull`s go along their business, flying about carelessly. But soon you want a seagull that can do something more, a seagull that can be your friend. You want a seagull that can play soccer! Instead of creating a new class that includes all the `Seagull` code all over again, you can start by declaring a `SoccerSeagull` that inherits from `Seagull`. This creates a parent-child relationship and makes `SoccerSeagull` exactly like `Seagull`. The `SoccerSeagull` inherits both the attributes (being 2 lbs) and the operations — squawking, flying, and so on — of its superclass, `Seagull`, without rewriting a line of code. Now you can add new functionality, like playing soccer, to the class.

In the file `ch4ex1.as`:

```
package {
  import com.actionscriptbible.ch4.Seagull;
  import com.actionscriptbible.ch4.SoccerSeagull;
  import flash.display.Sprite;

  public class ch4ex1 extends Sprite {
    public function ch4ex1() {
      var normalGull:Seagull = new Seagull();
      normalGull.fly();
      normalGull.squawk();

      var crazyGull:SoccerSeagull = new SoccerSeagull();
      crazyGull.fly();
      crazyGull.squawk();
      crazyGull.playSoccer();
    }
  }
}
```

In the file `com/actionscriptbible/ch4/Seagull.as`:

```
package com.actionscriptbible.ch4 {
  public class Seagull {
    public var weight:Number = 2;
    public function Seagull():void {
      trace("A new seagull appears.");
    }
    public function squawk():void {
      trace("The seagull says 'SQUAAA!'");
    }
    public function fly():void {
      trace("The seagull flies in a lazy circle");
    }
```

```
      public function eat():void {
        trace("The seagull digs its beak into the sand, pulls up a tiny, \
    struggling crab, and swallows it in one gulp.");
        weight += 0.5;
      }
    }
  }
```

In the file com/actionscriptbible/ch4/SoccerSeagull.as:

```
  package com.actionscriptbible.ch4 {
    public class SoccerSeagull extends Seagull {
      public function playSoccer():void {
        trace("Incredibly, the seagull produces a miniature soccer ball and \
    deftly kicks it into a goal.");
      }
    }
  }
```

This code provides the classes for the Seagull class and the SoccerSeagull classes. Note that you use the extends keyword to declare the parent class that the class will inherit from, because a child class can *extend* the abilities of its superclass. The example also shows that no code is required for the SoccerSeagull subclass to inherit the functions squawk(), fly(), and eat() from its superclass.

Additionally, you can use inheritance to *override* some behavior of the parent. For example, if your seagulls are creating a lot of noise, you might want to modify their squawking behavior. If you create a QuietSeagull that inherits from Seagull, the QuietSeagull will do everything a Seagull does, including squawk. But you can specify attributes of the superclass that you want to modify and provide new attributes in their place. So you could create a squawking behavior in QuietSeagull that squawks more quietly or even modify it to make no noise!

Through all this, the original Seagull is untouched. You can still have original, classic Seagulls running about among their bizarre offspring. Inheriting a class does not in any way alter the class; rather uses the parent class as a starting point for a child class. Moreover, subclasses are still considered their superclass: a SoccerSeagull is still a Seagull. It doesn't stop being a Seagull just because it can play soccer, and it can still do all the things a Seagull always could. The example shows crazyGull, the SoccerSeagull instance, flying and eating just like a normal Seagull does.

Inheritance is a concept applicable to classes, not objects. So a SoccerSeagull subclass can inherit from the Seagull class, but a Seagull object, once created, can't change its class. In the same way, the properties of an object that are defined or changed at runtime won't be part of the subclass's instances. Put another way, if a seagull injured its foot, it might be affected for the rest of its life, but its children will not inherit that trait or any other trait the parents acquired in the course of life. In the following example, the parent gains some weight, but you see that the new object inherits the default weight provided by the Seagull template rather than the runtime value of its parent.

```
  package {
    import com.actionscriptbible.Example;
    import com.actionscriptbible.ch4.Seagull;
    import com.actionscriptbible.ch4.SoccerSeagull;
```
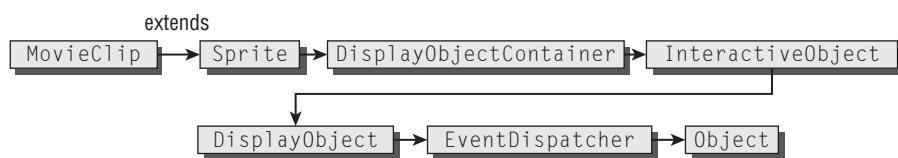
```
public class ch4ex2 extends Example {
  public function ch4ex2() {
    var gull:Seagull = new Seagull();
    gull.eat();
    gull.eat();
    gull.eat();
    gull.eat();
    trace("The parent seagull's weight is", gull.weight); //4

    var childGull:SoccerSeagull = new SoccerSeagull();
    trace("The child seagull's weight is", childGull.weight); //2
  }
}
}
```

Living things use a variety of reproductive practices, transferring their traits to their children in different ways. Humans have two parents and inherit their characteristics from a combination of both parents' traits. More simple organisms like viruses make a direct copy of their DNA or RNA so that their children inherit from only one parent. Inheritance in ActionScript 3.0 is this latter kind, called single inheritance. A class can inherit from only one class. In fact, if you don't specify a class to inherit from, your class extends the Object class, so all classes extend exactly one class (except Object itself). With single inheritance, you can draw a simple tree of classes, and you can trace any class back to Object with a simple list of classes. Things would get a lot more confusing if one class could inherit from multiple classes.

The whole list of classes that takes a class back to the root class, as shown in Figure 4-3, is called a class's *inheritance chain*.

FIGURE 4-3

Inheritance chain of the MovieClip class

Structuring Code with Inheritance

The MovieClip class is a truly complex class, with several sets of duties and almost 100 properties and methods. Important classes like these would quickly become a mess if it weren't for inheritance. Remember: one purpose of classes is isolating responsibility. With inheritance, you can ensure that those responsibilities are limited to the most specific possible interpretation of the class. All the more general responsibilities can be defined in a more general superclass.

Take the Seagull class, for example. Nowhere in its code did you have to define that it can lay an egg, that it has a beak and feathers, that it has two feet. These things are all true, of course, but they are really properties of birds. The Seagull class concerns itself with the specific responsibilities of a seagull: the way in which it flies, the kinds of food it eats, and its call. If you were to complete this example, you might have Seagull extend Bird, Bird extend Vertebrate, and Vertebrate

extend `Animal`. Each class would be responsible for only its own specific set of traits. Your inheritance chain might look something like Figure 4-4.
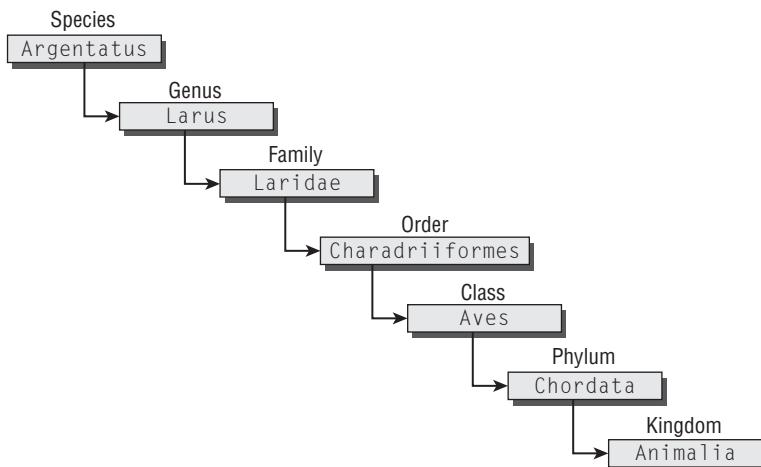
Inheritance chain of a Seagull class

Does this idea sound familiar? You already have an inheritance-based system for classifying the living things around you, in which the most specific traits are described in a species, and the most generic traits are described in a kingdom. The real scientific classification for the herring gull I have been describing looks like Figure 4-5, just in Latin instead of monospace.

Scientific classification of a herring gull (*Larus argentatus*)

The most specific classification is the species, and the most generic is the kingdom. Usually the scientific classification is written from general to specific, but here it is diagrammed to be like an inheritance chain, with the class or animal in question appearing first, and revealing its more generic classifications.

A goal of your object oriented designs should be to limit each class to one responsibility: managing itself. This sounds like circular reasoning, but really it's just an object-centric viewpoint. If you can create just the kinds of objects that are needed, they should be able to accomplish everything you desire. You can use inheritance to accomplish this. A `MovieClip`, as you will see in Chapter 14, "Visual Programming with the Display List," is a `Sprite` with a timeline added; a `Sprite` is just a `DisplayObjectContainer` that can contain multimedia content, be drawn upon and dragged, and so on. Using inheritance to create a family of classes that build on each other, you can make your code more modular, easier to modify, and simpler. One class, one responsibility.

Inheritance, Types, Polymorphism, and You

Inheritance gives rise to another important technique in the object oriented toolbox: polymorphism. This technique is used to write decoupled code — code which minimizes its dependence on other code — enabling you to add functionality to your programs without the heartbreaking process of tearing apart code that's already written, tested, and tuned. The name *polymorphism* makes this property sound terribly complex, but armed with an understanding of inheritance, it is easily understood.

Polymorphism is the practice of treating subclasses as the classes they inherit from. You know that a SoccerSeagull is still a Seagull and that an AcetyleneFlameToaster is still a Toaster. Both of these subclasses are able to do everything that their parents can, a fact guaranteed by the mechanism of inheritance.

A typical usage of polymorphism is when you have a family of related objects that behave slightly differently and some functionality that doesn't care how they differ. This happens all the time. You don't need to know much about objects to store them, keep track of them, and arrange them. You don't need to know if a seagull plays soccer or not to see it fly. If all you want is toast, you probably don't care about how the toaster does its job. In all of these cases, you could manipulate the objects as if they were more general types than they really might be.

The following example shows a seagull circus in action. This function, which could be part of a seagull trainer's repertoire, doesn't really care what kind of seagull is performing, since the trainer knows that all seagulls can fly and squawk.

```
package {
  import flash.display.Sprite;
  import com.actionscriptbible.ch4.*;

  public class ch4ex3 extends Sprite {
    public function ch4ex3() {
      var normalGullBob:Seagull = new Seagull();
      var normalGullJoe:Seagull = new Seagull();
      var sportyGullFrank:Seagull = new SoccerSeagull();
      var strangeGullNed:Seagull = new ThreeLeggedSeagull();

      seagullShow(normalGullJoe, strangeGullNed, sportyGullFrank);
    }
    private function seagullShow(gull1:Seagull,
                                 gull2:Seagull,
                                 gull3:Seagull):void {
      gull1.fly();
      gull2.squawk();
      gull3.fly();

      gull1.squawk();
      gull2.fly();
      gull3.squawk();

      //get a treat for performing
      gull1.eat();
      gull2.eat();
      gull3.eat();
    }
  }
}
```

The act can go on with any of the performers you created, since they are all seagulls. By creating the method `seagullShow()` so that its parameters are typed as instances of `Seagull`, you gain the ability to add all kinds of new `Seagulls` which can still participate in this sideshow code without any modification. In fact, the example goes further here by putting the specialized seagulls into variables that hold the general type.

You will learn more about polymorphism when you look at interfaces later in this chapter, and you learn how to get specific types out of generic variables when I discuss type manipulation.

Inheritance vs. Composition

Inheritance is not the only technique you can use to augment classes with new abilities. Let's say you have a `Kitchen` class that does all the good things a kitchen does, like letting you fry, sauté, broil, parboil, and bake things, and wash and store dishes and cutlery. If your goal is to give people a place to live, you're going to need a lot more than that. So you can extend the `Kitchen` to add some sofas, beds, bathtubs, tables, mirrors, sinks, parking, and maybe a fireplace. Now you have a specialized kind of `Kitchen`, say, a `CrowdedKitchen`, that people can live in. You can see immediately what's wrong with this example. A house isn't a special kind of kitchen; it's a structure with many kinds of rooms, including a kitchen. If you start adding these things to the `Kitchen` class, it really stops being what you might think of as a `Kitchen`.

Instead, let's build a `House` class that contains a `Kitchen`. When you have this `House`, you can continue to have all the convenience of a `Kitchen` just by accessing the `Kitchen` object inside it. You can continue to build out the `House` with lots of other rooms, adding in a `Bedroom`, `Bathroom`, `LivingRoom`, and `Garage`. Now you've created a composite class, a class that provides all the convenience of a `Kitchen` by *composition* instead of by *inheritance*. Furthermore, this way of structuring the house follows the object oriented design principle of "one class, one responsibility." Following is a `House` class with a kitchen, living room, and any number of bedrooms:

```
package {
  public class House {
    public var kitchen:Kitchen;
    public var livingroom:LivingRoom;
    public var bedrooms:Array;

    public function House(numberOfBeds:int) {
      kitchen = new Kitchen();
      livingroom = new LivingRoom();
      bedrooms = new Array();
      for (var i:int = 0; i < numberOfBeds; i++) {
        bedrooms[i] = new Bedroom();
      }
    }
  }
}
```

There's another way you can use composition to build up additional functionality. When the president of a company asks a manager to get something done, the manager tells her subordinates to do whatever parts of that task are necessary, and then she presents the final result. The `Manager` needs to coordinate these actions with `Employees` that she oversees, and as far as the outside world is concerned, the `Manager` indeed provides the result of the work of her `Employees`. But in the house example, the `House` simply provides access to the other classes it composed. If you want to cook something in the `House`, you get access to the `Kitchen` directly and use methods of `Kitchen`. Contrast this with the `Manager`, which *delegates* the activities to its constituent `Employee` objects.

This method also provides better encapsulation, as the president of the company should not have to deal with the employees directly.

```
package {
  import com.actionscriptbible.Example;

  public class ch4ex5 extends Example {
    public function ch4ex5() {
      runCompany();
    }
    protected function runCompany():void {
      //The manager has ten employees.
      var manager:Manager = new Manager(10);
      //When you ask the manager to do work,
      var completedWork:int = manager.delegateWork();
      //You get the work of ten people.
      trace(completedWork); //10
    }
  }
}

class Employee {
  public function doWork():int {
    //Each employee does one unit of work.
    return 1;
  }
}

class Manager {
  protected var employees:Array;

  public function Manager(numberOfEmployees:int) {
    employees = new Array();
    for (var i:int = 0; i < numberOfEmployees; i++) {
      employees[i] = new Employee();
    }
  }
  public function delegateWork():int {
    var totalWork:int = 0;
    for each (var employee:Employee in employees) {
      totalWork += employee.doWork();
    }
    return totalWork;
  }
}
```

In the example, even though the `Manager` has `Employee` objects, you ask the `Manager` instance to do the work, not the `Employees`. The `Manager` utilizes its `Employees` to do the work without your needing to know how or even needing to know that they are. This is a good example of composition, as well as encapsulation.

Whether a class delegates responsibilities like a `Manager` or simply aggregates objects like a `House`, the class uses composition to add functionality instead of inheritance. The typical rule of thumb is this: inheritance is an *is-a* relationship, and composition is a *has-a* relationship. An extra-loud seagull *is a* seagull, and a house *has a* kitchen.

This distinction is usually fairly clear, unmistakably so in these simple examples. But now and then there are ambiguous situations. The classic example is when you are creating a visual object. You can see your `IceCreamCone` class, but does that mean that your class should inherit from `Sprite`, or should it merely contain a `Sprite`? (A `Sprite` is a visual object in ActionScript 3.0. You can learn more about it in Chapter 14.) In Flash Professional, the recommended way to get your movie up and running is to assign a class to the root document. This class, when you define it, extends the `DisplayObject` of your choice. (For more information on `DisplayObjects`, please see Chapter 14.)

Similarly, in the Flex framework, your application starts with a class that must extend `Application`, also a view class. So there's some official sanction, if you need it, to create view classes that extend display classes rather than composing them. However, you should still apply the test to your own code. Does your `IceCreamCone` *have* a visual component, or *is it* a visual component? If this class, in addition to looking like an ice cream cone, has other functionality, like storing a flavor, accepting toppings, and melting by degrees, it really *isn't* a display object but *has* one. Following closely to the guideline, you might create `IceCreamCone` without extending `DisplayObject` but compose the visual object — the view — along with it.

Preventing Inheritance

In rare cases, you may want to ensure that the class you create is never extended. The class might contain functionality that would be dangerous to modify. Another class might subclass yours, modify the behavior of one or more methods, and pass itself off as your class, since a subclass is also the type of its superclass. Of course, as the programmer, you ultimately have control over all the code going into your program, but when precompiled libraries are involved and projects become large, things can get out of control.

Even if malicious code is not your concern, you can use this ability to signify that the class just should not be extended, that the class performs its sole duty, and there is no possible extra duty it can conceivably take on. It can be your signal that the class works best with composition instead of inheritance.

For whatever reason you choose, you can make sure that nobody inherits your class by declaring it *final*. To declare a final class, simply add the `final` keyword to the class declaration.

Note
The `final` keyword can come either before or after the access control attribute (in this case `public`). By convention, it is placed after. ∎

```
package {
  public final class Encrypter {
    public function encrypt(sourceText:String, key:String):String {
      //secure encryption algorithm goes here
    }
  }
}
```

Now if someone passes you an `Encrypter`, you can be sure that it's actually *this* `Encrypter`, and not a subclass like this impostor:

```
package {
  public class MaliciousEncrypter extends Encrypter {
    override public function encrypt(sourceText:String, key:String):String {
```

```
        return sourceText; //mwa ha ha!
      }
    }
  }
```

The `MaliciousEncrypter` class can pass itself off as an `Encrypter` because it is a subclass of `Encrypter`. However, it overrides the encryption algorithm to do nothing, defeating the security of an encryption class. However, as long as the `Encrypter` class is marked `final`, adding the `MaliciousEncrypter` class to your project will result in a compiler error. Extending `Encrypter` is forbidden as soon as you declare it `final`.

Extension is also prohibited on several classes that are built into the Flash Player API because their functionality is set in stone. For example, `flash.system.Security` works the way it is designed to work, and you should not be able to create your own implementation of the security system in Flash Player. Thus, `Security` is final.

Access Control Attributes

Access control attributes, or visibility modifiers, are keywords that are used with a piece of code to define who can access it. Most of the time, you use these with properties and methods of classes. In fact, all the examples in this chapter thus far have already used them.

By having granular control over the visibility of different parts of your classes, you can determine your own level of encapsulation, giving you control over how classes look to the outside world. At the same time, you can use functions and achieve good code reuse, while making sure those functions are only used when you say they are. The available access control attributes are listed in Table 4-1.

TABLE 4-1

Access Control Attributes

Attribute	Meaning
public	Available to all code
private	Available only within the same class
protected	Available within the same class and subclasses
internal	Available to code within the same package
A custom namespace	Available where the custom namespace is opened or when the identifier is prefixed with the namespace

You've already seen the `public` attribute used before all your class definitions. By making a class inside a package public, you enable the rest of the code in your program to use that class. In almost every case, you are writing a class so that everyone can use it, so putting `public` before `class` can become second nature.

Note

Runnable examples in this book make extensive use of classes outside the package block to keep the examples in one file; these class definitions aren't declared public (because only one publicly accessible item can be put in any file). This is abnormal. In code you write for projects, you will almost always use multiple files each with one public class. ■

Public and Private

The `public` and `private` attributes are used to modify methods and properties of classes. Private variables and methods typically allow a class to do internal work necessary to carry out the public requests made of it. Public variables and methods define what the class does. When you're designing a class, you should think carefully about what your class should be able to do and make those responsibilities public methods. For example, a toaster class should be able to take in some bread and change the bread into toast.

```
package com.actionscriptbible.ch4 {
  public class HeatedResistorToaster extends Toaster {
    public function HeatedResistorToaster() {
      // initialize the toaster
    }

    public function toast(pieceOfBread:Bread):Bread {
      //do whatever we need to toast the bread
      return pieceOfBread;
    }
  }
}
```

The `toast()` method should be public because this is the core responsibility of a `HeatedResistorToaster`. Other code in your program should be able to use a toaster to toast bread. But you still need to add in the steps necessary that are going to toast that bread. The private variables and methods constitute the internal mechanics of the toaster that users of the toaster should not have to deal with.

```
package com.actionscriptbible.ch4 {
  public class HeatedResistorToaster extends Toaster {
    private var remainingToastSec:Number;
    private var insertedBread:Bread;
    private var resistor:Resistor;

    public function HeatedResistorToaster() {
      resistor = new Resistor();
      remainingToastSec = 0;
    }

    public function toast(pieceOfBread:Bread):Bread {
      insertedBread = pieceOfBread;
      startTimer(10);
      startCurrent(0.5);
```

71

```
      while (remainingToastSec > 0) {
        tick();
        remainingToastSec--;
      }
      stopTimer();
      stopCurrent();
      insertedBread = null;
      return pieceOfBread;
    }

    private function startTimer(toastDuration:Number):void  {
      remainingToastSec = toastDuration;
    }

    private function startCurrent(currentAmps:Number):void {
      resistor.current = currentAmps;
    }

    private function tick():void {
      resistor.tick();
      insertedBread.toastiness += resistor.heat * 0.55667;
    }

    private function stopTimer():void {
      trace("The toaster sounds off a DING!");
    }

    private function stopCurrent():void {
      resistor.current = 0;
    }
  }
}
```

Now you've added a full algorithm to toast the bread. All these private functions and variables make up the way that heated resistor toasters make toast. First look at the properties you added.

The remainingToastSec variable represents the toaster's timer. When you put the bread in and start toasting, you shouldn't be able to fiddle with the time left, so remainingToastSec is private. The insertedBread variable keeps track of the bread inside the toaster to operate on. You can't mess with the bread inside the toaster after you plunge it in, so this is private. The resistor is the wires or coils running inside the toaster that heat up and toast the bread. Messing with those could be dangerous! All these variables are private because they represent either the state of the Toaster object or internal parts that are used in its business.

You also added several private methods to the class. These embody the steps that the machinery of the toaster follows to make bread into toast. If you were able to call startCurrent() from the outside, for example, you could force this toaster to unexpectedly become very hot, or if you were able to call startTimer() from the outside, you could interrupt the toasting process.

The public properties and methods of a class make up its public interface. All nonpublic parts of the class are involved with its implementation. This is how access control enables you to practice encapsulation. The public interface of a HeatedResistorToaster is just a toast() method, but internally you see that it has a state and operations that help it do the toasting.

Protected

A drawback of private items is that they are not inherited by subclasses. So if the `Toaster` class had a private property `finish` that described its paint job, the `HeatedResistorToaster` would not be able to share it. You would have to add it again to the `HeatedResistorToaster` definition and to any other class that derives from `Toaster`.

In the file `com/actionscriptbible/ch4/Toaster.as`:

```
package com.actionscriptbible.ch4 {
  public class Toaster {
    private var finish:String = "Reflective Silver";
  }
}
```

In the file `com/actionscriptbible/ch4/HeatedResistorToaster.as`:

```
package com.actionscriptbible.ch4 {
  public class HeatedResistorToaster extends Toaster {
    public function HeatedResistorToaster() {
      trace(finish); //Compiler error! We don't have a finish.
    }
  }
}
```

When you use inheritance to extend classes and you want items in the superclass to be available to the subclass but not to the outside world, use the `protected` attribute. A property or method declared as protected in a class will be available to any class that inherits from it.

To get the `finish` property to be available to `Toaster`'s subclasses, you change its definition to `protected`. In the file `com/actionscriptbible/ch4/Toaster.as`:

```
package com.actionscriptbible.ch4 {
  public class Toaster {
    protected var finish:String = "Reflective Silver";
  }
}
```

In the file `com/actionscriptbible/ch4/HeatedResistorToaster.as`:

```
package com.actionscriptbible.ch4 {
  public class HeatedResistorToaster extends Toaster {
    public function HeatedResistorToaster() {
      trace(finish); //Reflective Silver
    }
  }
}
```

It's almost always a good idea to keep your classes open for extension. If, after you wrap up and deliver your toasting program, you find that you need more toasting power, you might decide to create a toaster class with multiple resistors. A quick way to do this is to extend the `HeatedResistorToaster` class and add a second resistor, modifying the `startCurrent()` and `stopCurrent()` methods to support two resistors. To do this, you need to go back and make those methods protected instead of private. Whether private or protected, those methods are not available to

classes outside the `Toaster` class hierarchy, so you haven't lost anything, but you needed to modify a class that you should have just been able to extend.

Unless you have a reason to hide a property or method from every class in the class hierarchy or you're sure the class will never be extended, you should use `protected` over `private`. Using protected variables makes code easy to extend, even if you don't foresee a reason to right now. Use `protected` to encourage extension and `final` to prohibit it.

Tip

The idea that you should be able to add on to your program without changing anything that's already written is a principle of object oriented design called the *Open-Closed Principle*. According to the principle, code you write should be open for extension but closed for modification. This recognizes that code that's done means code that's been thought about, written, and tested. A lot of work has gone into it. It also recognizes that you should be able to add functionality to programs after you consider them "done," because even if you hate doing it, you always need to. Composition and inheritance are ways to add more functionality without touching code that already exists; you should favor this to modifying completed, tested code. ■

Internal

The `internal` attribute is tied to the concept of packages. Frequently you have classes that need to work closely together and sometimes have privileged access to each other: access that should not generally be open but is acceptable within a certain group of classes. For example, in the toaster example, you had to modify the `toastiness` of the bread directly. But why would you need a toaster at all if that attribute were publicly available? If `toastiness` were public, a `Bicycle` or a `Seagull` could go about toasting bread as if it were its business. This is definitely not desirable, but you have to have some way to change the properties of the bread. In this case, it might be a good idea to keep all the kitchenware classes in a `kitchen` package, including the `Bread` class, and to make `toastiness` an internal property. Now only kitchenware is allowed to modify the `toastiness` of bread.

In the file `com/actionscriptbible/ch4/kitchen/Bread.as`:

```
package com.actionscriptbible.ch4.kitchen {
  public class Bread {
    internal var toastiness:Number = 0;
  }
}
```

Now you attempt to toast the bed from a nonkitchenware object. In the file `com/actionscriptbible/ch4/bedroom/Pillow.as`:

```
package com.actionscriptbible.ch4.bedroom {
  import com.actionscriptbible.ch4.kitchen.Bread;

  public class Pillow {
    public function Pillow() {
      var b:Bread = new Bread();
      b.toastiness = 20; //Compiler error! You can't do this!
    }
  }
}
```

Structuring packages in terms of modules or subsystems can help you add another level of organization above classes. When the `internal` attribute is used, you can hide implementation details of subsystems from other subsystems, creating a kind of package-level encapsulation.

Custom Access Control with Namespaces
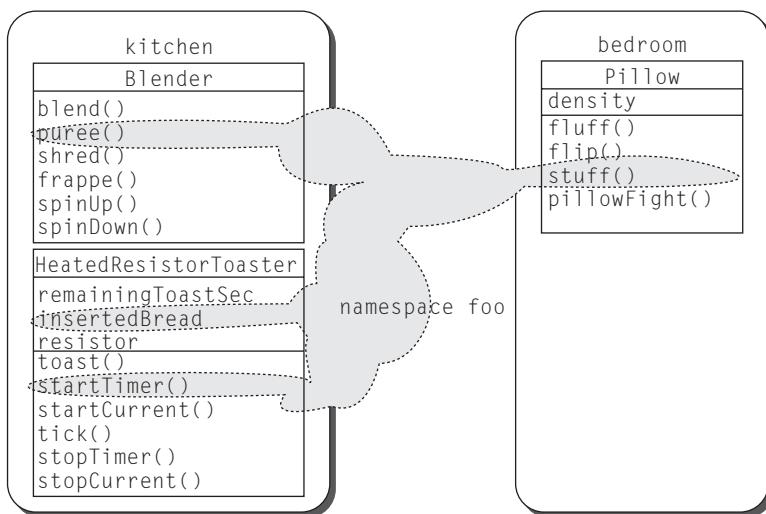
When you want even more control over who can access items, you can create a namespace. A *namespace* is a context in which names are unique, so you create namespaces every time you write a new package. In this context, namespaces collect arbitrary sets of functions, variables, and constants into one context regardless of where they are defined. Figure 4-6 shows a regular, controlled structure created by packages like `kitchen` and `bedroom`, and classes like `Pillow`, `Blender`, and `HeatedResistorToaster`. However, a namespace can contain whatever items you want. It lets you break out of the order imposed by packages and classes and overrule limitations of public, private, and protected access. In Figure 4-6, regardless of their natural grouping, seemingly unrelated methods like `Blender.puree()` and `Pillow.stuff()` can be grouped by creating a custom namespace like `foo`.

Cross-Reference

This type of namespace relates to which other classes can access a particular property or method. You may also use namespaces for XML. For more information about XML namespaces, see Chapter 11, "XML and E4X." ■

FIGURE 4-6

A namespace collecting items in the same context regardless of their surrounding organization

There are three steps to gaining and sharing special access to items with namespaces. First, declare a new namespace. Second, determine what items belong in that namespace. For those items, use the namespace's name as a visibility modifier instead of `public`, `private`, `protected`, or `internal`. Third, in code that needs special access to the namespaced item, open the namespace that you created.

Creating your own namespace and using it as a visibility modifier for an item allows anyone who opens the namespace to access the item. It's like locking the item in a cupboard but leaving the key taped to it: it conveys the information that the item you are qualifying is meant only for certain uses, and it requires you to be explicit that you are using it but doesn't make it impossible to use.

The Flex framework is an excellent example of the use of namespaces. Collaborating classes use many of the properties and methods of classes in the framework to help arrange them. However, these low-level functions should not be exposed to end users of the Flex components, so they are specified in the `mx_internal` namespace. When writing code that needs low-level control over components, you can dig deeper into the component classes simply by opening the `mx_internal` namespace and gaining access to the properties and methods.

You can create a new namespace with the `namespace` keyword. The `namespace` item, too, must be able to be located from code that wishes to either put itself in the namespace or use code from the namespace. However, the namespace is usually used to allow communication between packages, so you would have to make the namespace itself public so that both packages can add their items to it.

```
package com.actionscriptbible.ch4 {
  public namespace myns = "http://actionscriptbible.com/myns";
}
```

In this example, you define a namespace, `myns`, in its own file, `com/actionscriptbible/ch4/myns.as`. Now all code in this project should be able to use this namespace after importing it. The URL assigned to the namespace is optional. It allows the compiler to make sure that the namespace is truly unique by checking it against a presumably unique URL. Like package names, this is just a convention, and the URL is never fetched, nor does it have to really exist or contain anything. Note that this is a namespace, not a variable. You don't include a type declaration, and you can't assign a value to it or access it like a variable in your code, though when you declare it you use the assignment operator (=) to associate it with a URL.

To add items to this namespace, you simply use the namespace identifier as the access control attribute. First you have to import the namespace from its location in another package. When it is in the same package or the default package, this step is unnecessary.

```
package com.actionscriptbible.ch4.package1 {
  import com.actionscriptbible.ch4.myns;

  public class Class1 {
    myns function sayHi():void {
      trace("Hi!");
    }
  }
}
```

Note

The custom namespace is an alternative to the public, private, protected, or internal namespaces. The namespaces cannot be used together.

Code in custom namespaces may also be unavailable to the rest of the class it is defined in, if that class does not open the namespace. For example, other methods in `Class1` **would not be able to call** `sayHi().` ∎

The following set of classes shows two ways to use methods defined in a namespace. In the file `com/actionscriptbible/ch4/package2/Class2.as`:

```
package com.actionscriptbible.ch4.package2 {
  import com.actionscriptbible.ch4.myns;
  import com.actionscriptbible.ch4.package1.Class1;

  public class Class2 {
    use namespace myns;

    public function Class2() {
      var class1:Class1 = new Class1();
      class1.sayHi(); //Hi!
    }
  }
}
```

This class from `package2` is able to call the `sayHi()` method from a class in `package1`. The `use namespace` directive is like an `import` statement for namespaces. After opening the namespace with `use namespace`, all code in the class will be able to use items from that namespace.

```
package com.actionscriptbible.ch4.package3 {
  import com.actionscriptbible.ch4.myns;
  import com.actionscriptbible.ch4.package1.Class1;

  public class Class3 {
    public function Class3() {
      var class1:Class1 = new Class1();
      class1.myns::sayHi(); //Hi!
      class1.sayHi(); //Compiler error
    }
  }
}
```

This class from `package3` is also able to use the `sayHi()` method from `Class1` without opening the namespace. To use items from a namespace on a per-case basis, qualify the item you want to access with a namespace followed by the name qualifier operator (`::`). You still have to import the namespace to be able to use it. Since the second attempted call to `sayHi()` doesn't qualify the namespace and the namespace isn't open, the compiler will keep you from accessing it.

Despite the detail given here, custom namespaces are infrequently needed. Like the Flex framework, namespaces might be best used for whole systems that have implementation details that should be hidden from most users when they are redistributed.

Methods and Constructors

Through the examples in this chapter, you have already been exposed to dozens of methods, so let's formalize what you've already learned.

A method is a function that belongs to an object. (You learned about functions in Chapter 3, "Methods and Functions.") Methods are used to carry out the responsibilities of the class they are declared in. They can take parameters and return values. Methods can be accessible to different sets of code based on the access modifier applied to them, and if none is specified, the default, `internal`, is used.

There is a special kind of method that you have seen in many of the examples in this chapter. A *constructor* is a special method that executes when you create a new instance of the class with the `new` keyword. Its purpose is to initialize all the values of the object, create or retrieve collaborating objects where necessary, and put the object in a state where any of its public methods can be called without further ado.

Constructors are declared mostly like normal methods, except that

- Constructors have the same name as the class they construct.

- There can be only one constructor per class. Omitting it is possible, in which case, the class will use an empty constructor.

- Constructors must use the `public` access control attribute.

- Constructors don't return anything and don't specify a return type. However, you can think of constructors as returning the newly created instance if it makes more sense that way, since the expression `new MyClass()` evaluates to the newly created instance of `MyClass`.

Caution

In ActionScript 3.0, the constructor must always be `public`. ■

Take a look at the constructor from the `HeatedResistorToaster` class, from an earlier example:

```
public function HeatedResistorToaster() {
    resistor = new Resistor();
    remainingToastSec = 0;
}
```

Note the lack of return type or `return` statement. The constructor sets the initial values for some of the state variables and creates an object (`resistor`) that the class uses to do the toasting. If any public methods were called on the `HeatedResistorToaster` before this executed, they would generate runtime errors. That's because all of them rely on the `resistor`, and the constructor would not have created it yet. Thankfully, you know that the constructor of a class runs as the class is instantiated, before you have the chance to call any methods on it. For any instance of any class, the constructor always runs first.

Constructors are allowed to take parameters like any other method, and those parameters are passed inside the parentheses in the `new` statement:

```
new SomeClass(12, "Helvetica", true);
```

There isn't much stylistic choice when it comes to constructors since they inherit the name from their class. However, constructors should be the first method in a class listing. Usually when you write classes, the properties are grouped at the top, followed by the constructor, followed by public methods and finally nonpublic methods.

Properties

Properties, or instance variables, are variables that are part of an object. They can be used to store attributes of an object or other objects it owns or is keeping track of. They can be defined like any other variable, and they have the same access control attributes as methods. Constants are special

properties that don't change values and are declared with the `const` keyword instead of `var`. Both constants and properties default to `internal` when no access control attribute is specified. Figure 4-7 illustrates the declaration of an instance variable.

FIGURE 4-7

A vivisected instance variable.

```
internal var toastiness:Number = 0;
```

```
  access
├─ control ─┤├─var or─┤├─identifier─┤├─ type ─┤├─ initial ─┤
  attribute       const                          value
```

The access control modifier, type, and initial assignment in a declaration like that in Figure 4-7 are optional. Like methods, properties default to `internal` when another access control modifier is not specified.

Just like variables, properties are *nouns* and *adjectives* and should be named as such. They can be singular or plural. They should have descriptive names and be typed in camel case. Properties should be declared at the top of a class definition, before the constructor.

The choice to use initial values or not can be one of personal style. Any assignment that can be made at the declaration time of the property may also be made inside the constructor of the class with no ill effect. If initial values are used at all, they are typically used for Boolean properties, constant numbers and strings, and other simple values.

Accessors

When the class is a simple data type, there is no shame in using public properties. For example, a `Point3D` class that just collects three values is a good use of public properties:

```
package {
  public class Point3D {
    public var x:Number;
    public var y:Number;
    public var z:Number;
  }
}
```

In general, however, public properties should be avoided in favor of *accessors*. Changing the property of an object by assigning directly to the property doesn't allow that object the opportunity to react to the change, and it shifts the burden of updating the object to every other method that might need to know if the value has changed. It doesn't give the object an opportunity to ensure the value assigned to its property is valid, so invalid input or code can easily trip up a perfectly working object.

Accessors fix this problem by allowing you to execute code when assignments are made to and when values are retrieved from an object. You can use two kinds of accessors in ActionScript 3.0: explicit and implicit.

Explicit accessors are normal functions that can retrieve and modify a property. There are no rules to explicit accessors, and they are not a special part of the language. They are typically implemented like this:

```
package com.actionscriptbible.ch4.accessors {
  public class BuildingExplicit {
    //the height of the building in feet
    private var _height:Number = 0;

    public function setHeight(newHeight:Number):void {
      if (!isNaN(newHeight) && newHeight >= 0 && isFinite(newHeight)) {
        _height = newHeight;
        updateFloors();
      } else {
        throw new RangeError("Height must be finite and positive");
      }
    }

    public function getHeight():Number {
      return _height;
    }

    private function updateFloors():void {
      //Make sure the number of floors
      //jives with the height of the building
    }
  }
}
```

The real, internal property is made to be private. One convention for handling properties with both a private version and public accessors is to name the private version of the property with a preceding underscore. Two normal methods are added to set and retrieve the property. These should be named setFoo() and getFoo() for the property named foo, and they are appropriately called the setter and the getter.

In this example, the setter ensures that the attempted assignment is valid and only proceeds if it is. If an invalid assignment is attempted, the class throws a runtime error. For more on error handling, see Chapter 24, "Errors and Exceptions." The setter also takes the opportunity to update the building's floors based on the new height. Otherwise, you always have to be on your feet, checking that the two are in sync. By handling that at the only point it might happen, you can eliminate the problem.

Another thing you can do with accessors is to work with *derived properties*: values that aren't stored but are calculated on demand, masquerading as normal properties. For example, you could add to the Building class an accessor for the height of the building in inches:

```
public function getHeightInches():Number {
    return getHeight() * 12;
}
```

When using explicit accessors, you must use method calls to manipulate the public properties of the class.

```
var b:BuildingExplicit = new BuildingExplicit();
b.setHeight(100);
trace("The building is", b.getHeight(), "feet tall");
```

The second way to use accessors enables you to use dot notation as if the properties were public, but still intercept the assignment to do whatever you require. *Implicit accessors* are a special language feature and use the keywords get and set before the property names to define accessor functions. In other words, just add a space between set and height in the example:

```
package com.actionscriptbible.ch4.accessors {
  public class BuildingImplicit {
    //the height of the building in feet
    private var _height:Number = 0;

    public function set height(newHeight:Number):void {
      if (!isNaN(newHeight) && newHeight >= 0 && isFinite(newHeight)) {
        _height = newHeight;
        updateFloors();
      } else {
        throw new RangeError("Height must be finite and positive");
      }
    }

    public function get height():Number {
      return _height;
    }

    private function updateFloors():void {
      //Make sure the number of floors
      //jives with the height of the building
    }

    public function get heightInches():Number {
      return height * 12;
    }
  }
}
```

To access the properties, you can use dot notation and still reap the benefits of the range checking code:

```
var b:BuildingImplicit = new BuildingImplicit();
b.height = 1000;
b.height = -12; //Runtime Error: Height must be finite and positive
```

You can also use implicit accessors to make a property read-only. Include a public implicit getter but omit the setter. The compiler catches any attempts at setting the property from outside the class. In the Building example, the property heightInInches is, in addition to being derived, read-only.

Avoid Side Effects

I just showed how to execute arbitrary code when a property assignment is made, but it's important that I issue a warning here. It's a good idea to use accessors to control access to properties and to ensure that a modified property has an immediate effect on anything that depends on it. However, when you call on an object to perform some task and in addition it performs something unrelated, this is known as a *side effect*, and although it is often convenient, you should generally avoid it. You should design your classes, methods, and properties so that they do what their names imply, and no more.

For example, say you wanted to repaint your building a new color to match its height, or create new elevator banks, every time the height is changed. These would be nice, but they make changes to the Building that should probably be calculated and made explicit. In fact, one could even argue that

adding floors is a side effect. Only change or update parts of the class that are absolutely dependent on the property that's changing. Methods as well as accessors can have side effects, so remember that setting values and performing actions should remain separate. It can be useful to use implicit accessors to beef up handling of values, but don't use it to confuse adjectives with verbs.

Self-Referential Code

In Chapter 2, "ActionScript 3.0 Language Basics," you learned about scope. Local scopes are created by blocks — code enclosed within curly braces — within which identifiers and names are unique. When you're writing methods in classes, several scopes are open at the same time, because the package, class, and method all have their own code blocks:

```
package com.actionscriptbible.ch4 {
  public class Camera {
    private var aperture:Number;
    private var shutterSpeed:Number;
    private var focalLength:Number;
    private var currentFrame:Number;
    public function takePhoto() {
      var image:Image = new Image();
      openShutter();
      accumulateImage(image, shutterSpeed);
      closeShutter();
      currentFrame++;
    }
  }
}
```

The open blocks are made more obvious by the fact that each one indents its contents further. Code inside the `takePhoto()` method can access variables defined in the method block like `image`, instance variables defined in the class like `shutterSpeed`, as well as any other items defined in the package, if there were any. This is because of the way ActionScript code searches up the scope chain from the most local open scope to the most global.

Because each block defines a namespace, you can easily have local variables inside a method with the same name as instance variables defined on the class level. Inside the method, the local variables take precedence, since the method block is the most local. Outside of the method, those variables don't even exist.

```
package com.actionscriptbible.ch4 {
  import com.actionscriptbible.Example;
  public class ScopeTest extends Example {
    private var a:String = "apricot";
    public function ScopeTest() {
      testA(); testB();
    }
    private function testA():void {
      var a:String = "acid";
      trace(a); //acid;
      trace(b); //ERROR: Access of undefined property b
      trace(this.a); //apricot
    }
```

```
      private function testB():void {
        var b:String = "banana pancakes";
        trace(a); //apricot
        trace(b); //banana pancakes
        this.finish();
      }
      private function finish():void {
        trace("done!");
      }
    }
  }
```

In `testA()` here, the local definition of a *shadows* the instance variable a. Any mentions of a evaluate to the local variable a. In `testB()`, there is no local variable a, so a evaluates to the instance variable a.

Inside a class, you can always reference the instance of the class itself by using `this`. Therefore, the `this` keyword can be used to specifically target instance variables that have been shadowed, as in the last line of `testA()`. In fact, any item defined at the class level can be referenced using this, whether it is shadowed or not. For instance, in `testB()` you call `this.finish()`, a method which is perfectly visible otherwise.

Some developers use `this` frequently in constructors that initialize instance variables:

```
package com.actionscriptbible.ch4 {
  public class Camera {
    private var aperture:Number;
    private var shutterSpeed:Number;
    private var focalLength:Number;
    private var currentFrame:Number;
    public function Camera(aperture:Number = 3.5,
                           shutterSpeed:Number = 1/400,
                           focalLength:Number = Number.POSITIVE_INFINITY) {
      this.aperture = aperture;
      this.shutterSpeed = shutterSpeed;
      this.focalLength = focalLength;
      this.currentFrame = 1;
    }
  }
}
```

Using Static Methods and Properties

To recap, instance methods and instance properties are items that are defined in an object's class and exist in the object. A `Bicycle` class might have a property for the number of gears in the bike.

```
package com.actionscriptbible.ch4 {
  public class Bicycle {
    private var _gears:Number;

    public function Bicycle(numberOfGears:Number) {
      this._gears = numberOfGears;
    }
```

```
      public function get gears():Number {
        return _gears;
      }
    }
  }
```

This example uses accessors to good effect, enabling you to create a new `Bicycle` with any number of gears but never allowing you to change the number of gears once it is created. It does this by assigning the number of gears in the constructor and providing no setter for the property.

The concept here is that an instance variable, like `gears`, belongs to the particular instance. Each instance of `Bicycle` can have its own number of gears.

```
var eighteenSpeed:Bicycle = new Bicycle(18);
var singleSpeed:Bicycle = new Bicycle(1);

trace(eighteenSpeed.gears); //18
trace(singleSpeed.gears); //1
```

There are other properties of a bicycle you might think of to make into instance variables, like the color of the bike, the size of the bike, or the kind of tires on the bike. Some properties, however, are part of the *definition* of a bicycle. All bicycles, by definition, have two wheels. These properties can be modeled by static variables.

Static Variables

ActionScript 3.0 enables you to define methods and properties that belong to the class, rather than their instances. Since the number of wheels is part of the definition of a bicycle rather than a property of the bicycle itself, you can add this property to the class rather than the instance. This is called a *class variable*, or a *static variable*.

```
package com.actionscriptbible.ch4 {
  public class Bicycle {
    public static var wheels:Number = 2;
    private var _gears:Number;

    public function Bicycle(numberOfGears:Number) {
      this._gears = numberOfGears;
    }
    public function get gears():Number {
      return _gears;
    }
  }
}
```

Note
The `static` keyword can be written before or after the access control attribute. The convention is to write it after the visibility modifier, as shown here. ■

By adding the `static` keyword, you are defining the variable `wheels` as part of the class instead of the instance. All `Bicycle` objects will now share the same `wheels` variable. Even if you create a fleet of a thousand bikes, all will share this class variable, and any changes to it will immediately affect all the objects.

Now that `wheels` is a static variable, you use it differently. It belongs to the class `Bicycle`, so you use dot notation to access it from the class, not the instance.

```
var singleSpeed:Bicycle = new Bicycle(1);
trace(singleSpeed.wheels); //Wrong! Compiler error.
trace(Bicycle.wheels); //Right! 2
```

All the normal access rules still apply, so you can only write `Bicycle.wheels` because `wheels` was defined as public. Remember that you can see the `Bicycle` class by virtue of the fact that it, too, is public.

Code that goes inside the class can access the `wheels` variable without writing out `Bicycle`
`.wheels`. Since it is in the same class that the static variable is defined in, the variable is in scope. However, many people think that it's good style to always reference static variables with their class names. This way, every time you reference a static variable it's quite clear that it's static and not an instance variable. Following, you add code to the constructor of your `Bicycle` that uses the static variable.

```
package com.actionscriptbible.ch4 {
  public class Bicycle {
    static public var wheels:Number = 2;
    private var _gears:Number;

    public function Bicycle(numberOfGears:Number) {
      this._gears = numberOfGears;
      for (var i:int = 0; i < Bicycle.wheels; i++){
        //Prepare a wheel.
      }
    }
    public function get gears():Number {
      return _gears;
    }
  }
}
```

Static Constants

When you think about it, since the number of wheels is part of the definition of a bicycle, the `wheels` variable should not only belong to the `Bicycle` class, it should be a constant. You can easily have static constants — constants that belong to the class instead of the instance.

```
package com.actionscriptbible.ch4 {
  public class Bicycle {
    static public const WHEELS:Number = 2;
    //and so on...
```

In fact, it turns out that most values that must remain fixed must also remain fixed across all instances of a class. This is not a rule, but a common case. Things like terminal velocity, the melting point of lead, and the number of letters in the alphabet are constants, but also constants that can remain constant across instances and which you would write as static. When a constant doesn't change between instances of a class, and when you need to be able to access a constant without constructing an instance of the class that owns it, use static constants.

Static constants also allow for some useful techniques that are worth mentioning here.

String comparisons and string parameters can be messy. Consider the following function that you might place in the `Bicycle` class:

```
package com.actionscriptbible.ch4 {
  public class Bicycle {
    public function performTrick(trickName:String):void {
      switch (trickName) {
        case "wheelie":
          //code goes here to wheelie
          break;
        case "bunnyhop":
          //code goes here to bunny hop
          break;
        case "stoppie":
          //code goes here to stoppie
          break;
      }
    }
  }
}
```

This performs the right kind of trick based on the name of the trick you pass it. The problem here is that if you misspell the trick accidentally, nothing happens, which is disappointing. You might end up scratching your head, wondering where your error is, without realizing that there's no error except the spelling of the trick.

```
var stuntBike:Bicycle = new Bicycle();
stuntBike.performTrick("wheely"); //nothing happens
```

However, if you replace all the strings with constants

```
package com.actionscriptbible.ch4 {
    public class Bicycle {
        public static const WHEELIE:String = "wheelie";
        public static const BUNNYHOP:String = "bunnyhop";
        public static const STOPPIE:String = "stoppie";

        public function performTrick(trickName:String):void {
            switch (trickName) {
                case Bicycle.WHEELIE:
                    //code goes here to wheelie
                    break;
                case Bicycle.BUNNYHOP:
                    //code goes here to bunny hop
                    break;
                case Bicycle.STOPPIE:
                    //code goes here to stoppie
                    break;
            }
        }
    }
}
```

the compiler catches that spelling mistake, since there is no WHEELY constant.

```
var stuntBike:Bicycle = new Bicycle();
stuntBike.performTrick(Bicycle.WHEELY); //compiler error
```

Tip

If you use a development environment with code hinting such as Flash Builder, you can use the auto-completion features to help you find the valid trick name interactively. ■

This practice is used heavily in the flash.events package but can be used any time you need to compare against a special string. You can do even more with this case, however. Really, you'd like a trick to be its own type, which can have the values of wheelie, bunnyhop, or stoppie. Right now it's a String with special values defined in the Bicycle class, where arguably they don't belong.

Enumerations

A custom type you create that can only have certain discrete values is called an *enumeration*. Action-Script 3.0 does not have built-in support for enumerations, but they're easy enough to build. You can create a Trick class that enumerates the kinds of tricks the bike supports:

```
package com.actionscriptbible.ch4 {
  public final class Trick {
    public static const WHEELIE:Trick = new Trick();
    public static const BUNNYHOP:Trick = new Trick();
    public static const STOPPIE:Trick = new Trick();
  }
}
```

This might seem odd at first, but WHEELIE is not only a static property of the Trick class, but an instance of Trick. By making the tricks instances of Trick and not just strings stored by Trick, you can type variables as Tricks. Now you can make sure that the parameter passed to the performTrick() function is a real Trick, not just some bogus string that may or may not be valid. All you have to do is change the signature of the function to accept a Trick parameter instead of a String parameter.

Enumeration classes are also ideal final classes. Because you're probably writing code based on the assumption that the enumeration contains only certain values, creating a subclass that adds values will prove a messy mistake. This is one of those cases in which modifying your original code is better than creating a subclass.

Static Methods

Variables and constants aren't the only code that can belong to a class. There can also be methods that operate independently of any instance. These static methods must not access any instance variables since they will be run by the class itself, possibly before the class is ever instantiated.

Static methods can be used when you want to run some code without actually creating the object. In the following example, you replace a normal constructor with a static method. You can use this method to create new instances of the object.

```
package com.actionscriptbible.ch4 {
  public class LimitedEdition {
    private static var editionsMade:int = 0;
    private static const MAX_EDITIONS:int = 20;

    private var serialNumber:int;

    public function LimitedEdition(serialNumber:int) {
      this.serialNumber = serialNumber;
    }

    public static function getOne():LimitedEdition {
      if (editionsMade++ < MAX_EDITIONS) {
        return new LimitedEdition(editionsMade);
      }
      return null;
    }

    public function toString():String {
      return "Limited Edition Object #" + serialNumber;
    }
  }
}
```

This object is a limited edition. Only 20 may be created, and they are indelibly branded with their serial number when they are created. A class variable and class constant keep track of the number of instances made already and the total number of instances that can be made. Because the number of instances created is kept inside the class, it is intransient, able to exist inside `LimitedEdition` regardless of whether there are any `LimitedEdition` objects in existence. After the target number of editions has been created, the static creation method refuses to return a new one.

Just like a class variable, you call class methods by using dot notation on the class name. The `getOne()` static method is a function you call on the class to return an instance and keep track of how many `LimitedEdition` instances have been created. In contrast, the `toString()` instance method is called on an instance to determine what your object will look like represented as a string:

```
for (var i:int = 0; i < 50; i++)
{
    var obj:LimitedEdition = LimitedEdition.getOne();
    trace(obj.toString());
}
/*
prints out:
Limited Edition Object #1
Limited Edition Object #2
Limited Edition Object #3
Limited Edition Object #4
Limited Edition Object #5
...
Limited Edition Object #19
Limited Edition Object #20
null
null
null
```

```
null
...
null
*/
```

This approach still has a weakness, though. Remember that one of the requirements of a constructor is that it's public. Therefore, you can't prevent the ability to create new instances by bypassing your static creator method and calling the constructor. This can be overcome by throwing an exception in the constructor. (You'll learn about exceptions in Chapter 24.) The important thing about this example is that static methods can be used when there is no instance of the class to speak of, and these methods can work with other static variables and methods.

Another way that static methods are frequently used is to create utility classes. Utility classes are classes that aren't meant to be instantiated; they exist to group some related functions, all of which are declared statically. Because of this, utility classes aren't really object oriented, and if you rely on them too much you might end up programming procedurally instead of object oriented.

A poster child for utility classes is the `Math` class, which you will explore in Chapter 7, "Numbers, Math, and Dates." `Math` provides a bunch of really useful mathematical functions all in one place, so it's a great utility. You would never create a new `Math` object, but you would definitely call its static method `Math.pow(2, 10)` to raise 2 to the 10th power or access the `Math.PI` static constant for the value of π.

You can create static implicit and explicit accessors as well. Simply use the `static` keyword with your method names, and you can create static properties that are derived or read-only. Keep in mind, however, that static methods — accessors included — can't call nonstatic methods of the class.

Overriding Behavior

Now that you've seen methods and access modifiers in action, allow me to return to inheritance for a moment. When using inheritance, sometimes you want to make the child class do something that the parent class does, but differently. For example, you might want to make a `QuietSeagull`, which makes a much softer sound when it squawks. The superclass, or base class, of `Seagull` already has a `squawk()` behavior, and by extending that class, your new subclass inherits that behavior.

Plain vanilla inheritance provides the new subclass with a starting point, which is all the public, internal, and protected properties and methods of the base class. But sometimes adding onto this starting point is not enough. Sometimes you have to change it.

You can *override* methods of a base class by using the `override` keyword. You can put this keyword before or after the access control attribute. Because you don't have access to them, you can't override any private methods from the superclass.

The `QuietSeagull` class might look something like this:

```
package com.actionscriptbible.ch4 {
  public class QuietSeagull extends Seagull {
    override public function squawk():void {
      //A very polite seagull does not squawk,
      //despite what its parents may do.
      trace("...");
    }
  }
}
```

Now if you create several different kinds of seagull and ask them all to squawk, you'll find that the original `Seagull`, and all classes that extend it without overriding `squawk()`, act the same. But the `QuietSeagull` marches to the beat of his own, quiet drum:

```
var normalGull:Seagull = new Seagull();
var sportyGull:Seagull = new SoccerSeagull();
var quietGull:Seagull = new QuietSeagull();
normalGull.squawk(); //The seagull says 'SQUAAA!'
sportyGull.squawk(); //The seagull says 'SQUAAA!'
quietGull.squawk(); //...
```

You can only override methods of a class. You can't override properties, like the seagull's weight. However, by using accessors you can get around this. Because accessors are just methods, you can override them — even implicit accessors — to return a different value in the subclass. For example, you can convert the `Seagull` base class to expose its `weight` as an accessor rather than a public property, and override that. The side benefit is that the seagull's weight can become read-only so that you can control the weight internally in response to eating and other behaviors but disallow external modification of weight.

In the file `com/actionscriptbible/ch4/Seagull.as`:

```
package com.actionscriptbible.ch4 {
  public class Seagull {
    public function get weight():Number {
      return 2;
    }

    public function squawk():void {
      trace("The seagull says 'SQUAAA!'");
    }
  }
}
```

In the file `com/actionscriptbible/ch4/HungrySeagull.as`:

```
package com.actionscriptbible.ch4 {
  public class HungrySeagull extends Seagull {
    override public function get weight():Number {
      return 1.5;
    }
  }
}
```

Note

Overriding methods is different from overloading methods. Overloading allows two methods to have the same name if they have different argument lists. You can't do this in ActionScript 3.0. Overridden methods have to have the same signature as the original method. In ActionScript 3.0, you can only override methods. ∎

Accessing the Superclass

When you override a function, you don't have to throw out everything that the base class did for you. You still have access to the nonprivate members and properties of the superclass, through the `super`

object. Use super to add on to an existing method or call functionality in the superclass that has been overridden by other overrides.

You can create a polite seagull that does everything a normal seagull does but also apologizes afterward. To make this work, you need to modify the main methods of a Seagull, but also use the way they already work. Rather than copy that code, you depend on the superclass's implementation. This also means that changes in the implementation of methods on the base class are reflected immediately in the subclasses.

```
package com.actionscriptbible.ch4 {
  public class PoliteSeagull extends Seagull {
    public function PoliteSeagull() {
      super();
      trace("It seems very polite.");
    }
    override public function squawk():void {
      super.squawk();
      trace("The shy gull covers his mouth in shame.");
    }

    override public function fly():void {
      super.fly();
      trace("The gull lands and apologizes for blocking out the sun.");
    }

    override public function eat():void {
      trace("The gull apologizes to the animal it's about to eat.");
      super.eat();
    }
  }
}
```

The polite seagull example shows that you can call the base class's methods, whether you call them before, after, or in the middle of the code you add.

```
var politeGull:Seagull = new PoliteSeagull();
//A new seagull appears
//It seems very polite.
politeGull.eat();
/*The gull apologizes to the animal it's about to eat.
The seagull digs its beak into the sand, pulls up a tiny struggling crab,
and swallows it in one gulp.*/
```

Constructors in subclasses are a bit special. Each constructor function, like each class, is unique. You don't actually override the superclass's constructor. However, it is always called. When accessing the superclass's constructor, instead of using the super object to call a superclass's method, you use the super() method. This method calls the superclass's constructor, no matter what the superclass is named. You must put this call somewhere in your constructor, because you don't really extend from your superclass unless you've given the superclass a chance to initialize itself. If you omit a call to super(), it is called at the beginning of the constructor. If you omit the constructor entirely, the default constructor runs, calling the superclass's constructor.

Designing Interfaces

When you extend a base class, you have the opportunity to override any of its methods and provide new behavior. You also can't help but inherit all the existing methods and variables that aren't declared private. Inheritance is useful for augmenting existing classes with new functionality, but it has its drawbacks.

Being forced to start off with all the methods and properties of the superclass can be a drawback. If you want a family of classes to perform the same kind task in radically different ways, it can be stifling to have to work around existing code.

You can work around this by having a relatively empty base class that your family of classes extend. But then you are presented with another problem: this base class is a perfectly valid, although entirely ineffectual, class. So the empty base class could be instantiated and used in your program, achieving nothing where something was undoubtedly expected.

You are also presented with the problem of multiple inheritance. Classes can only extend one other class. This enables you to create a class hierarchy in the shape of a tree and not an acid-induced spider web. Single inheritance makes sense when you're modeling your classes after whole, fully thought-out objects: An object can't be both a house and a hamburger, but an object *can* perform two different kinds of tasks. A tree belongs to both a class of objects that provides shelter and a class of objects that can yield food. It would be nice to be able to use the tree as a provider of food when you need food and use it as a provider of shelter when you need shelter.

What you're working toward is a completely abstract item that classes can implement in different ways and that you can combine more than one of in one class. An *interface* fulfills these requirements. Interfaces are representations of a certain ability that don't say anything about how the ability will be carried out but describe the ability. Interfaces are not classes, and they can't be instantiated. They contain no code. Put another way, interfaces are contracts. When you choose to implement an interface, it's entirely up to you how to do it, but you must meet its specifications and fulfill its contract. If inheritance is an "is a" relationship and composition is a "has a" relationship, implementing an interface is a "can do" relationship.

This can be best explained in an example. This interface would be stored in a file called `com/actionscriptbible/ch4/IPowerable.as`:

```
package com.actionscriptbible.ch4 {
  public interface IPowerable {
    function turnOn(volts:Number):void;
    function turnOff():void;
    function getPowerUse():Number;
  }
}
```

Note

In ActionScript 3.0, type annotations — additions to the name of an item to indicate its usage or type — are not required, and they are not typically used. However, it remains customary to prefix interfaces with the letter I as in `IPowerable`. ∎

No access control attribute is required on methods defined in an interface, because interfaces define the public interface to an object. The methods must by definition be public. Interfaces may not specify properties, but they may specify explicit or implicit accessor functions.

The interface IPowerable creates a promise of some kind of behavior. It specifies how things that can be powered will work with the outside world. The interface specifies a contract that the compiler guarantees will be followed in classes that choose to be IPowerable.

Classes that implement this interface can do much, much more than these three methods, but they are required to implement these three methods.

Clearly, the set of objects that can draw power can have diverse behaviors. A lamp, for example, draws electricity and produces light:

```
package com.actionscriptbible.ch4 {
  public class Lamp implements IPowerable {
    private var watts:Number;
    private var isOn:Boolean;

    public function Lamp(wattage:Number) {
      watts = wattage;
      isOn = false;
    }

    public function turnOn(volts:Number):void {
      isOn = true;
      trace("it gets brighter!");
    }

    public function turnOff():void {
      isOn = false;
      trace("it gets darker.");
    }

    public function getPowerUse():Number {
      return (isOn)? watts : 0;
    }
  }
}
```

The Lamp class fulfills the contract set out by the IPowerable interface by declaring the three methods in the interface with identical method signatures. The names of the parameters don't matter for a method's signature, but the types, order, and number of those parameters do. The implements keyword is used to make the class adhere to the interfaces that are specified. A class can both extend another class and implement an interface at once, but the extends section must come before the implements section.

You can create objects that implement the interface in different ways. The toaster, for example, doesn't actually start drawing current until you put in a piece of toast and start toasting. However, as far as IPowerable is concerned, as long as it can do turnOn(), turnOff(), and getPowerUse(), the object is an IPowerable.

```
package com.actionscriptbible.ch4 {
  public class Toaster implements IPowerable {
    private var isOn:Boolean;
    private var isToasting:Boolean;
```

```
public function turnOff():void {
  isOn = false;
  isToasting = false;
}

public function getPowerUse():Number {
  if (isToasting) {
    return 100;
  } else {
    return 0;
  }
}

public function turnOn(volts:Number):void {
  isOn = true;
}

public function toast(pieceOfBread:Bread):Bread {
  if (!isOn) {
    trace("nothing happens");
    return pieceOfBread;
  }
  trace("your bread gets toasty");
  isToasting = true;
  pieceOfBread.toastiness += 10;
  return pieceOfBread;
}
  }
}
```

To use the `Lamp` and `Toaster` classes, you create and manipulate them like you normally would. But certain other classes just don't care what kind of appliance an object is, as long as it implements the `IPowerable` interface. Just like an interface is an agreement about how to use objects that implement it, the electrical system, too, is an interface. It says (in the United States) that as long as you insert a plug shaped like two prongs, you'll get 120 volts of current alternating at 60 Hz, and if the plug has a rounded pin below the prongs, you'll get a grounded connection. This is an agreement that would definitely fry everything you own if either appliance makers or electric companies decided to deviate from it. But because this interface is in place, you can take any electrical appliance you can think of and get electricity as easily and safely as sliding in a plug. Little agreements like this really do make the world go around, like the hundreds of protocols in use on the internet today, our expectations about how to push buttons and tap keys, and the common way you expect to use all kinds of vehicles. Interfaces allow systems to work together based on certain expectations of each other and not on their actual implementation.

Given this, you should be able to build a power strip that can power all your appliances at once. The power strip doesn't care what you plug into it as long as what you plug into it is `IPowerable`. `Lamps` and `Toasters` may come from completely different families of classes, but with interfaces, you can treat any object in terms of what it *agrees to do*, rather than what it is.

```
package com.actionscriptbible.ch4 {
  public class PowerStrip implements IPowerable {
    private var appliances:Array;
```

```
        public function PowerStrip() {
          appliances = new Array();
        }

        public function addAppliance(appliance:IPowerable):void {
          appliances.push(appliance);
        }

        public function turnOn(volts:Number):void {
          for each (var appliance:IPowerable in appliances) {
            appliance.turnOn(volts);
          }
        }

        public function turnOff():void {
          for each (var appliance:IPowerable in appliances) {
            appliance.turnOff();
          }
        }

        public function getPowerUse():Number {
          var powerDraw:Number = 0;
          for each (var appliance:IPowerable in appliances) {
            powerDraw += appliance.getPowerUse();
          }
          return powerDraw;
        }
      }
    }
```

Furthermore, this power strip should be powerable itself, so you can encapsulate all the appliances in one object. As far as the electrical system is concerned, the appliances in your house are one big appliance. This is great object oriented stuff: you can aggregate many objects into one and utilize that single object, abstracting away the details of its constituents. Isn't it more convenient in real life to turn off a power strip than to individually turn off all the things plugged into it?

Notice that in all the for..each loops and in the addAppliance() function, you use IPowerable as a type. As long as you can agree that the object implements IPowerable and limit yourself to the abilities that IPowerable specifically guarantees, there can't be any problem. This is what I mean by programming in terms of abilities instead of classes.

Now create a bunch of appliances and then test the power strip.

```
    var powerStrip:PowerStrip = new PowerStrip();
    var heater:Heater = new Heater(10);
    var toaster:Toaster = new Toaster();
    var readingLamp:Lamp = new Lamp(25);
    var overheadLamp:Lamp = new Lamp(60);
    powerStrip.addAppliance(heater);
    powerStrip.addAppliance(toaster);
    powerStrip.addAppliance(readingLamp);
    powerStrip.addAppliance(overheadLamp);
    toaster.toast(new Bread()); //nothing happens
    powerStrip.turnOn(120);
```

```
//it gets warmer! it gets brighter! it gets brighter!
trace(powerStrip.getPowerUse()); //1285
toaster.toast(new Bread()); //your bread gets toasty
trace(powerStrip.getPowerUse()); //1385
powerStrip.turnOff();
//it gets cold. it gets darker. it gets darker.
```

Note that although it does adhere to the `IPowerable` interface, the `toaster` variable still holds a `Toaster` reference, and it can still use methods specific to `Toaster` like `toast()`. You don't lose anything by implementing the interface.

A class can implement multiple interfaces, like the earlier tree example. To implement multiple interfaces, just separate the interface names with commas, so the tree example might look like this:

```
public class Tree extends Plant implements IShelterProvider, IFoodProvider
```

Note
Interface names often end in -able since they describe an ability. However, sticking to this nomenclature can often end in awkward interface names, and I recommend that you drop it when it doesn't make sense. ■

Interfaces can also extend other interfaces. For example, you might use this to create a category of objects that can be subscribed to as an event source in addition to the categorical ability. You can achieve this by extending the built-in `IEventDispatcher` interface. See more about event sources in Part IV, "Event-Driven Programming."

The use of interfaces is encouraged by object oriented designers because it keeps down dependencies between concrete classes. The more dependencies you create, the less opportunity you leave for yourself to change your program without impacting existing code, and the less maintainable your code becomes as the web of interdependencies becomes exponentially thicker. As I've mentioned, interfaces allow you to address objects not in terms of what they are but in terms of the minimum you need them to do, and this can give you micro control of how you deal with objects. Because objects can implement many interfaces, they can be one thing to one class and another thing to another class, as necessity dictates.

A principal rule of object oriented design is "program to an interface, not an implementation." The usage of "interface" here does not necessarily mean `interface`; it can also mean a superclass or whatever is the most generic type that gives you the abilities you need at that time. Implementations change, but interfaces should not: we have laptops now that people wouldn't believe 10 years ago, but a time-traveler could take them back in time and demonstrate them perfectly, because the interface for drawing power has not changed, nor has the idea of typing on a keyboard and watching a display.

Manipulating Types

The type system enables you to refer to an object by its own class, by any of its superclasses, by any interface it implements, or by a combination of these. All these potential types for an object are valid because, by being more specific, it must implement a superset of all the more general types' functionality. A `Square` is still a `Rectangle` because it is a specific kind of `Rectangle`, and it is a `Shape` because it is a specific kind of `Shape`. It might also be an `IPointCollection` if it can provide a set of points that comprise its vertices. The point is that these aren't conversions: the class really *is* all the things that it extends.

Because you can refer to a class by many names, you need ways to get between these types. You should be able to test at runtime if an object is an instance of a given class or implements a particular interface. You should be able to convert a specific type to a general type, and you should be able to attempt to convert general types to specific types. You should even be allowed to attempt to convert types to unrelated types.

Type Compatibility and Coercion

Some types can be converted without explicit action. Since an instance of a class is also an instance of its superclass, you can convert it to the more general type without writing any explicit code.

```
var square:Rectangle = new Square();
```

Again, this isn't really a conversion at all, but semantics. These types are compatible. A `Square` is a `Rectangle`.

In addition, ActionScript 3.0 knows how to make certain unrelated type conversions for you and will do so without asking. This is called *coercion*, or *implicit type conversion*. ActionScript 3.0 knows how to convert anything to a `Boolean`, for example, making the following kind of code tricks possible:

```
var array:Array = ["hello", "world"];
var obj:Object;
//Pull a value off the front of the array, set obj to that value,
//and check if obj converts to true or false. Stop the loop when
//the value converts to false. Null is returned by shift() when
//there are no more values in the array. Null converts to false.
while (obj = array.shift()) {
    trace(obj);
}
//hello
//world

var str:String = "";
//Set a default string if the string doesn't exist or is empty.
//Both empty strings and null values convert to false.
//Negating false gives you true so the conditional runs if
//the string is empty, giving it a default value instead.
if (!str) str = "Default";
trace(str); //Default

for (var i:int = 1; i < 4; i++)
{
    //Trace out the number (i) and whether it is odd or even.
    //i % 2 is the remainder when you divide it by 2, always
    //a whole number. Any number converts to true except 0,
    //which converts to false. So when the remainder after
    //dividing by 2 is 0, the conditional is false.
    trace(i, (i % 2)? "odd" : "even");
}
//1 odd
//2 even
//3 odd
```

The first trick shows that any object is `true` but `null` is `false`. The second trick shows that an empty string is `false`. The third shows that an integer is `false` if it's `0` and `true` otherwise.

Similarly, ActionScript 3.0 uses an object's `toString()` method to automatically convert any object to a `String`, using the `Object` class's `toString()` method if no other class up the hierarchy defines it.

Other implicit type conversions are available to ActionScript 3.0 base types, but these are discussed in the chapters having to do with their types. For example, `XML` can convert to a `String` automatically when used in a string context.
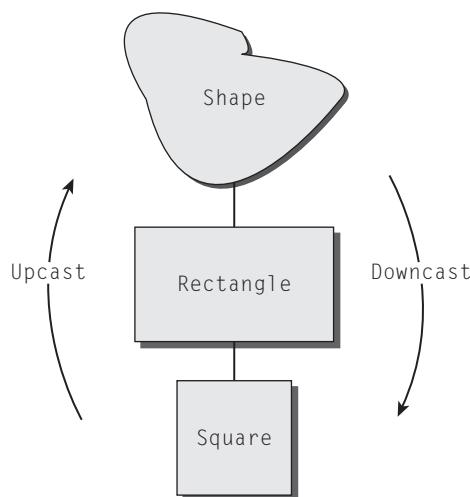
Explicit Type Conversion

You can also induce type conversion. ActionScript 3.0 provides two kinds of checked cast that enable you to attempt to convert from one type to another. A *cast* attempts to fit a type into another type. If they are compatible, the conversion will continue, and the expression will result in the new type. The types of cast differ in how they act when the conversion fails.

Another bit of terminology is the *direction* of a cast. An upcast is a type conversion *up* the inheritance chain, toward the more general. This kind of cast is always successful, and, as you have seen, can be achieved implicitly, without a cast operation. A downcast is a type conversion *down* the inheritance chain, toward the more specific. Since you can't be guaranteed that a particular `Rectangle` is a `Square`, if you try to cast it without looking, the cast might fail. Figure 4-8 shows how this works.

FIGURE 4-8

Upcasting and downcasting

I will demonstrate these casts with a downcast since it is likely to be unsuccessful, and you can see how the types of cast differ when the casts fail. The first kind of cast you can perform is safer and

preferable in most cases. To cast a variable to a type, surround the name of the instance with parentheses and put the type name before it, as if you were calling its constructor but without the `new` operator:

```
var someShape:Shape = getRandomShape(); //get some unknown shape
Square(someShape); //treat it as a Square
```

In this example, presume for the sake of argument that you have obtained a shape, but you don't know what kind of shape it is. This can happen when a method returns a general class rather than a specific one. This is represented with the method call `getRandomShape()`.

If the conversion is not successful — in this case, if `someShape` was actually a `Rectangle` with a different width and height — this cast will throw a `TypeError`. To learn more about how to handle these errors and why they are your friend, see Chapter 24.

To capture the result of this cast, you probably want to store the result of the cast in a variable of the type you cast it to:

```
var mySquare:Square = Square(someShape);
```

The second kind of cast does not throw an error if the cast fails; rather, the cast evaluates to `null`. An error is usually better, because as you test your program, you are immediately stopped where the cast fails, and not later, when this unexpected `null` causes some other error elsewhere and becomes a mysterious bug. That said, this kind of cast is achieved with the `as` operator, which treats the object on the left of the operator as if it were the class on the right.

```
var mySquare:Square = someShape as Square;
```

One common application of casting is when retrieving objects from a collection. Other than `Vector`, collections such as `Array` store their objects untyped. When you retrieve values from these collections, they are returned as `Object`s, since ActionScript can make no guarantees as to the contents of the array. So whenever you must iterate over items from a collection, you can cast these items into the class you know they originally were. Because they come out as `Object`, this is always going to be a downcast unless you were storing `Object` instances.

Another benefit of the constructor-style cast is that it takes advantage of several intelligent conversion functions, which are cleverly disguised as cast operators. The top level of ActionScript 3.0 includes, among other top-level functions like `trace()` and `isNaN()`, several functions that, when applied, have the same syntax as a cast. These functions behave just like a cast, too, but instead of failing to convert unrelated types, they perform an intelligent conversion. For example, the function `XML()` looks just like a cast to XML but converts XML text in a `String` into an actual `XML` object, whereas a cast would fail.

```
var myXml:XML = XML("<root><party><time>Now</time>\
<location>Here</location></party></root>");
trace(myXml.party[0].time); //Now
```

Because these top-level functions act just like a cast (with more intelligence) and masquerade as cast operators, now that you know they exist you can blissfully ignore them. Keep using the constructor-style cast, and you will benefit automatically from a few conversion functions.

Determining Types

To determine if an object is compatible with a class, you can use the `is` operator. This is a binary operator, not a function, so the syntax for using it is:

```
//get a shape from somewhere in the program
var someShape:Shape = getRandomShape();
if (someShape is Circle) {
    (someShape as Circle).circumference;
}
```

Here, only circles have circumferences, so if you can determine that the shape is a `Circle`, you can safely retrieve its circumference. Because the `circumference` property is only defined on instances of `Circle`, even though you know the `someShape` instance is in reality a `Circle`, you must convert its type to `Circle` before accessing `Circle` properties.

Also note that because you checked that the instance was a `Circle` type, there was no danger in performing the downcast, as you just checked its type compatibility in the line above it.

It is important to note that `is` determines the compatibility of a type, not the exact type of the instance.

```
var s:Shape = new Square();
trace(s is Square); //true
trace(s is Rectangle); //true
```

There is actually a great deal more you can do with the type of objects, including determining the exact type of instances and retrieving class references from the name of a class alone, but these topics, generally called *reflection*, are advanced techniques not covered in this chapter. You'll see the latter kind of reflection when you retrieve class definitions from an externally loaded SWF in Chapter 27, "Networking Basics and Flash Player Security."

Creating Dynamic Classes

The last kind of class is saved for the end because it should be used rarely and with care. Dynamic classes are classes whose properties and methods can be changed, added, and removed at runtime. This works against the type system and many object oriented principles, but it can be convenient for some needs.

You can create a dynamic class by adding the keyword `dynamic` to the class definition.

```
public dynamic class ShapeShifter
```

For a discussion of the benefits and properties of dynamic classes, see Chapter 10, covering `Object`, everyone's favorite dynamic class. For most cases in which you need a dynamic class, you might be satisfied with using an `Object` instance instead of an instance of a custom dynamic class.

Summary

- Classes are the templates for objects, the building blocks of your program.
- Objects have both data and operations.
- Classes should encapsulate their implementations and provide a simple interface.
- Classes are placed in packages, which structure and allow unique names for classes.
- Objects have a type, which is the class they are an instance of.
- All variables are objects — even numbers and Booleans.
- Objects have methods and properties, and classes can have static methods and static variables.
- You can use `public`, `private`, `protected`, `internal`, and custom namespaces to encapsulate and allow access to your objects.
- Classes can inherit from other classes, automatically inheriting public and protected properties and methods.
- Classes can only inherit from one other class, so all the classes can be drawn in a tree, and each class can have an inheritance chain up to the top of the tree.
- Classes should be closed for modification and open for extension.
- The public interface of a class is all of its public methods and properties and is how the class appears to the outside world.
- Interfaces define a contract that classes can choose to follow to provide some ability.
- You can use interfaces as types so that you can decouple your code from implementations.
- Types can be inspected, automatically coerced, or explicitly cast.

Validating Your Program

No matter how careful you are when writing code, you will almost certainly run into one or two errors. Don't worry; it happens to everyone, including myself every day. Being able to see error messages is a real blessing. Errors are your friends, in a way. Your goal should always be to strive for error-free code, but error messages are the best way to tell when there's something wrong that you may not have noticed. Think of it like a spell-checker. Compiling and running your code and checking for errors frequently is the best way to catch problems before they multiply.

Flash Player has not only a robust system for handling errors but support for interactive debuggers, both of while I'll cover in detail in Part V, "Error Handling." However, since you should be starting to play with the example code that you'll see more and more of, it's a good time to stop and talk about some of the kinds of bugs you're likely to run into and what to do about them.

Introducing Errors

Errors are messages generated by the compiler or runtime that let you know something has gone wrong.

Tip

A feature in the AS3 compiler allows you to set whether you want the compiler to interpret your code using *Strict mode* or not. Strict mode, the default setting, allows you to compile your code using more rigorous compile-time type checking. It checks to see that all classes, methods, and properties are valid before allowing the program to compile — hence, it's more strict. This makes debugging much easier; however, it also prevents you from dynamically accessing members of nondynamic objects. Non-Strict mode is much more like previous versions of ActionScript, which did not have strict compile-time type checking.

All examples in this book should be compiled with Strict mode enabled unless otherwise noted. ∎

Compile-Time Errors vs. Runtime Errors

You'll deal with two basic types of errors when writing programs in ActionScript 3.0: compile-time errors and runtime errors.

Compile-Time Errors

Compile-time errors or compilation errors occur when you attempt to compile your SWF file. These types of errors usually indicate a syntactic mistake, a missing `import` statement, or a reference to an item that does not exist. You must fix all compilation errors before the compiler can create an executable SWF. If you run a SWF before fixing compilation errors, you'll be running either an incomplete SWF or a version that was generated before you introduced the error. Either situation is sure to be confusing, so don't ignore compilation errors!

Runtime Errors

A runtime error is one that occurs during the execution of your SWF file — while the program is running. These usually arise from a problem that occurs after your program begins, such as a reference to an object that has been removed. Runtime errors can cause unpredictable results in your program. Some may cause the program to crash; others may go unnoticed. Either way, it's important to address any errors you have in your program.

Tip

To see runtime errors from your web browser, you need to install the Debug version of the Flash Player. This is installed along with all Adobe-packaged development environments by default; it is always available at `http://www.adobe.com/support/flashplayer/downloads.html`**.** ∎

Warnings

Sometimes the compiler issues a warning. Warnings concern problems that won't prevent your code from compiling but should be fixed for the sake of clean, readable, consistent code. A common example is when you forget to add a data type to a variable or function return. It's a good idea to heed any warnings generated and fix them as though they were errors.

Getting Feedback from Flash Professional and Flash Builder

Flash Professional and Flash Builder differ slightly when identifying errors. This should get you started with whichever software you use.

Note

Both Flash Professional and Flash Builder have Debugging modes that offer more advanced troubleshooting tools. These are covered in Chapter 25, "Using the AVM2 Debugger and Profiler." ∎

Debugging in Flash Professional

In Flash Professional, the output panel (for trace statements and runtime errors) is brought up automatically when you test your movie. If there are compile errors, a Compiler Errors panel also comes up.
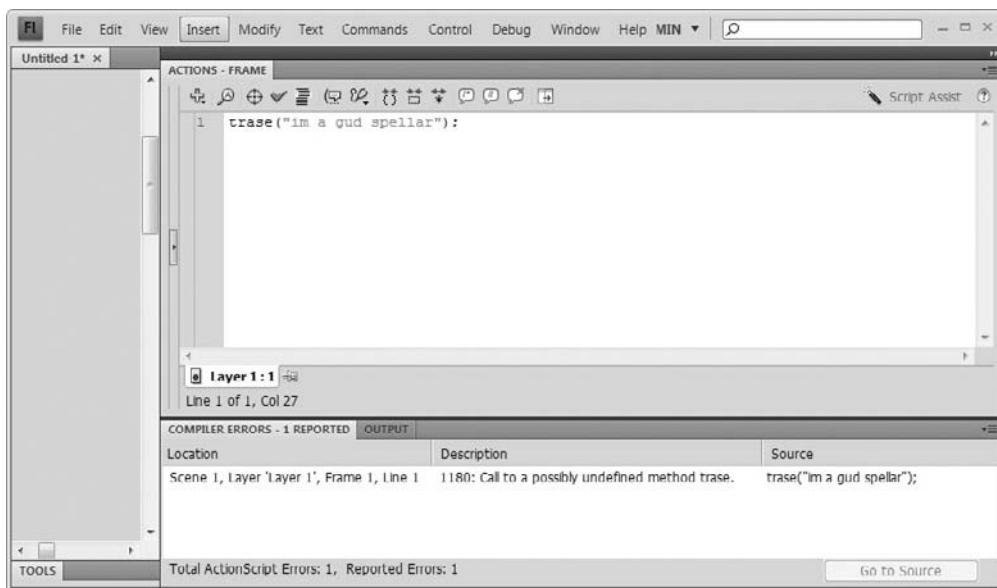
I'll use this code snippet to test the Compiler Errors panel. There's an obvious problem with this call to `trace()`:

```
trase("im a gud spellar");
```

Compiling shows the Compiler Errors panel with one error reported, as shown in Figure 5-1. The panel shows the location of the error either in an AS file or in a scene, layer, and frame in an FLA file along with the line number. If you highlight the error and click the Go to Source button in the bottom right, the cursor jumps to that location in the code. There's also a Description column with the error code and description of the error, and a Source column showing the offending line of code.

FIGURE 5-1

Flash Professional's Compiler Errors panel shows that there is no method called "trase."

In addition to the Compiler Errors panel is the Output panel. This shows trace and logging statements as well as errors that occur at runtime. Running this code
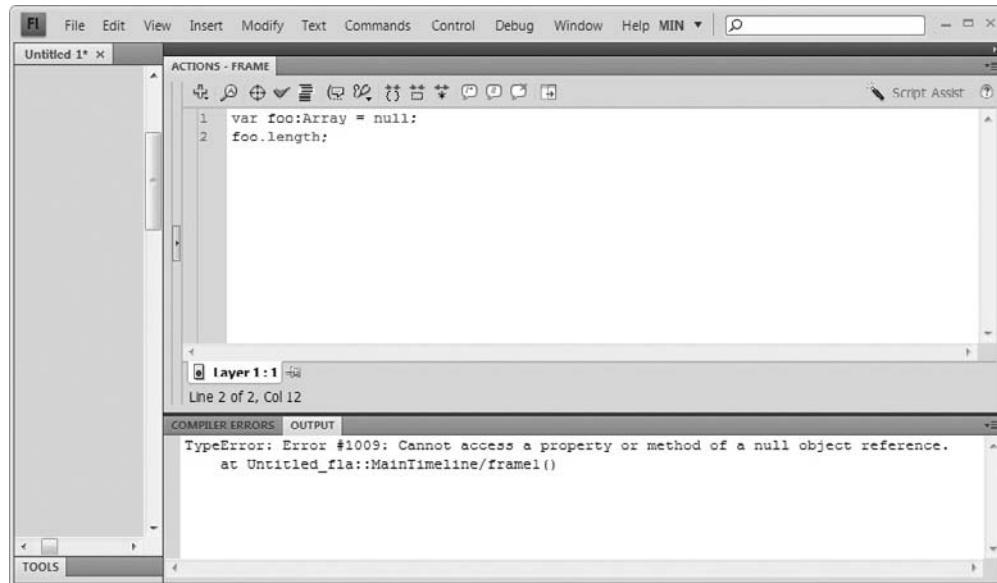
```
var foo:Array = null;
foo.length;
```

produces the result shown in Figure 5-2. Again, this output shows the call stack that was executing at the time of the error. Code that executes from the timeline will show the frame number as a function in the call stack.

FIGURE 5-2

Flash Professional's Output panel

Debugging in Flash Builder

One major advantage of using Flash Builder is its ability to notify you of problems in your code as they are written. Whenever you build your source code, markers appear next to line numbers that contain compile-time errors. Hovering over these markers or opening the Problems view shows you more detailed information about the problem, as shown in Figure 5-3. You can click on the red reference markers next to the scroll bar on the right of your code or double-click on an item in the Problems view to jump to the line of code with the error.

Warnings are reported in a similar way, but they use a yellow icon instead of red.

Tip

If Flash Builder is set to automatically build, you will see errors and warnings every time you save a file, as long as that file is an active application or is used by an active application. You can toggle this feature under the Project ⇨ Build Automatically menu item. ■

Similar to Flash Professional's Output panel, Flash Builder also has a Console view, shown in Figure 5-4, that shows traces, logs, and errors when debugging.

Flash Builder showing the error line marker and its counterpart, the Problems view
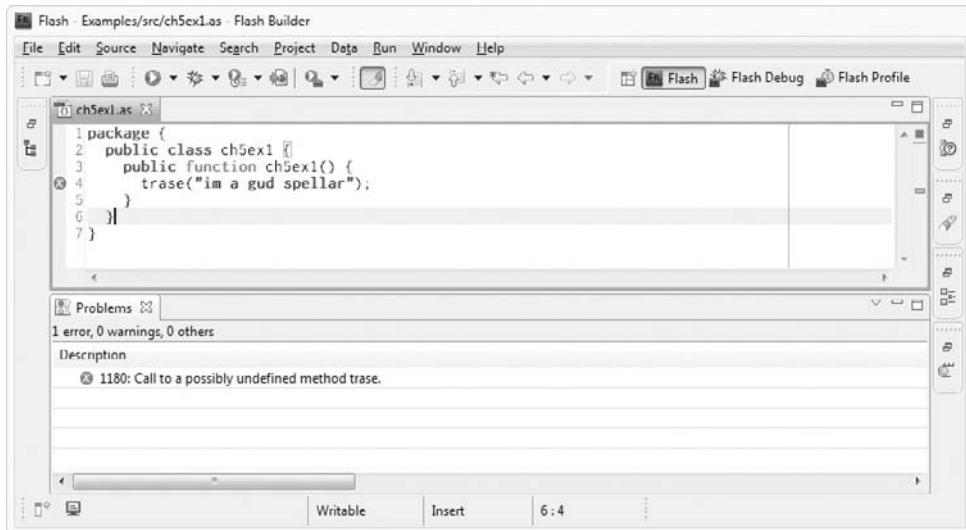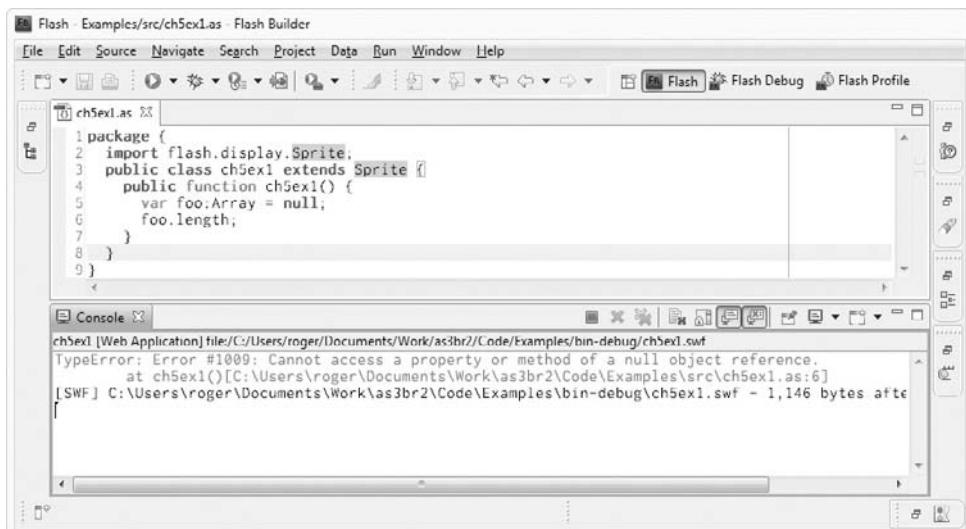
Flash Builder showing a runtime error in the Console view

Fixing Errors

When a compile-time error is reported in a particular section of the code, a line number and filename are given for reference. This is the best place to start when trying to fix the problems in your program. Go back to the line number where the error was found and look for typos or other mistakes that could have caused the error.

Runtime errors encountered in a debug version of Flash Player trace out the current *call stack*, a list of all the functions that are in the process of being executed at the time of the failure. Studying the call stack allows you to track back to the original function or functions that caused the problem. For example, consider the following code snippet, which causes a `TypeError` because `nullSprite` has no value:

```
package {
  import flash.display.Sprite;
  public class ch5ex1 extends Sprite {
    public function ch5ex1() {
      var nullSprite:Sprite = null;
      updateLayout(nullSprite);
    }
    public function updateLayout(sprite:Sprite):void {
      moveSprite(sprite);
    }
    public function moveSprite(sprite:Sprite):void {
      sprite.x = 100;
    }
  }
}
```

This code produces the following error:

```
TypeError: Error #1009: Cannot access a property or method of a null
object reference.
at ch5ex1/moveSprite()
at ch5ex1/updateLayout()
at ch5ex1()
```

As you can see, both `ch5ex1()` and `updateLayout()` make calls to other functions within their function block. As such, they appear in the preceding call stack. This can help tremendously to find the original source of the problem. (In this case, the `null` assignment in the constructor was the issue.)

Tip

You may see several compiler errors at once; sometimes one error can be the cause for another. Don't jump around when fixing errors. Generally speaking, it's best to start from the first error on the list, fix it, save your files, and recompile. Many times, this resolves more than one problem down the line. ∎

Table 5-1 is a list of the more common errors that you are likely to encounter — especially if you're just starting out with ActionScript 3.0 programming.

A List of Commonly Found Errors and Their Remedies

Error Description	Possible Causes	Possible Solutions
TypeError: Error #1009: Cannot access a property or method of a null object reference.	Caused when an attempt is made to access properties or methods to an object that was never assigned a value or has been deliberately set to null. This may be a shock to pre-3.0 ActionScript developers because previous versions did not treat this as an error.	Ensure that you properly define any objects whose methods or properties you need to access. Or you can add a check to your code to validate the object before calling methods on it. For example: `if (myArray) myArray.pop();`
C1100: Assignment within conditional. Did you mean == instead of =?	This warning appears when an assignment (for example, a=b) is made within a conditional statement such as `if`.	This is a common typo for new developers. Because the assignment (=) and equality (==) operators are so similar, they are often confused. Make sure you're using the right one for the job. `a = b` means ''Set a to equal b'' `a == b` means ''Is a equal to b?''
T1180: Call to a possibly undefined method _____. Or 1119: Access of possibly undefined property _____ through a reference with static type _____. Or 1120: Access of undefined property _____.	These errors can appear when you attempt to call a function, get or set a variable, or access an object that does not exist. For example: `var myArray:Array = new Array(); myArray.washMyCar();`	Typos are a common cause of these errors. Check through your code to make sure everything is spelled correctly. Remember that AS3 is case-sensitive, so `foo`, `Foo`, and `FOO` are three different things. Make sure your variables are properly declared using a `var` statement before you attempt to use them. In AS2, the `var` was optional, but it's required in AS3. Finally, make sure you're not trying to access a member of a class that does not exist. If necessary, check the documentation for the offending class to see if the property or method is valid.

TABLE 5-1   *(continued)*

Error Description	Possible Causes	Possible Solutions
1046: Type was not found or was not a compile-time constant: _____.	The class or interface you're trying to use couldn't be found. It might have been referenced incorrectly or might not exist at all.	This error is similar to the three I just mentioned. Check your code for typos around class names. Also, make sure you include `import` statements for any classes that aren't automatically imported. In AS3, even some of the built-in classes such as `flash .display.Sprite` need to be imported. Finally, check to make sure the filename and the name of the class are the same.
1067: Implicit coercion of a value of type _____ to an unrelated type _____. Or 1136: Incorrect number of arguments. Expected _____.	A call to a method was attempted with the wrong number of arguments (parameters) or with the wrong data type for one of the arguments.	If a function defines required arguments, the arguments passed to the function when it is called must match the function signature. That is, you must provide the correct number of arguments each with the data type that the function is expecting. This is a stricter rule than in previous versions of ActionScript. If you're not sure what arguments to use, check the documentation for the function you're trying to call.

Summary

- Errors happen all the time even for advanced programmers. Errors reported by Flash Player and the compiler are your best tool in making your code work better.

- Fix reported errors starting with the first one in the list and working your way down. Usually, fixing the first error makes it easier to fix the rest.

- Use the line numbers and the call stack provided with the errors to help identify the area of your code that is causing the problem.

- As you gain experience, you will become more familiar with common programming errors and will be able to catch them before they occur.

- For more info on errors and debugging, check out Part V of this book.

Part II

Core ActionScript 3.0 Data Types

Text, Strings, and Characters

Text in ActionScript is represented as a *string*, as in "the book consisted entirely of a long string of incomprehensible characters." A string can be any piece of textual data from a single letter to a word to the entirety of Wikipedia. Strings don't have to contain legible prose: they might also contain or be exclusively made of numbers or symbols. Strings can also handle multiple languages. Strings are an essential building block of virtually all programs, so ActionScript 3.0 provides several methods for working with, manipulating, searching, and converting strings.

FEATURED CLASS

`String`

Working with String Literals

Of course, because they're so indispensable, you've already been exposed to strings before this chapter. Most of what you've seen so far has been string literals — pieces of text surrounded by quotation marks added directly into the code. But as you learned in Chapter 2, "ActionScript 3.0 Language Basics," everything in ActionScript 3.0 is an object, including strings. When you type a string like `"Hello, world!"` you're actually creating an instance of the `String` class. The `String` class offers a load of extra functionality beyond the capability to store text.

Tip

Although technically there is a difference between a string created with a `new String()` statement and a string created using a string literal, the compiler converts string literals into string objects automatically. Therefore, the following code is perfectly valid:

```
"banana".length; // 6
```

Using Escaped Characters

String literals use quotes to tell the compiler that the text contained within them should be interpreted as a string. Both single and double quotes are used, and in some cases they can be used interchangeably. In other cases, you'll need to know their special properties to choose the right combination of quotes. Let's look at how they behave differently. Say you want to store a piece of text that contains quotes:

```
var porky:String = 'Porky says "That's all folks!"';
```

The compiler expects you to enclose string literals in matching types of quotes. In other words, whichever type of quote mark you use to begin the literal will be the type that the compiler looks for to denote the end of that literal. You can take advantage of this fact to put quotes inside literals: if you need to use single quotes in the string, you can delineate the literal with double quotes. If you need to use double quotes, you can sneak those in between single quotes.

In this example, I've started and ended with a single quote, and that lets me get away with double quotes inside the string. However, there's still a problem with it. The compiler will find an error in this code because the string is interpreted as `'Porky says "That'`. Once the compiler sees the single quote after `That`, it decides: End of string. Time to move on and keep reading code. But then, confronted with `s all folks!"'`, it has no idea what to do. In cases like this, you need to use *escaped characters*.

Escaped characters are special pseudo-characters that the compiler replaces with real characters that may be difficult to type or would be illegal to type in code. Escape characters are preceded by a backslash. The escape characters for double and single quotes are `\"` and `\'` respectively. The other escape sequences you can use are shown in Table 6-1. Here's an example:

```
var bugs:String = 'Bugs says \"What\'s up, doc?\"';
trace(bugs); //Bugs says "What's up, doc?"
```

When you use the escaped version of characters, they are immune to being treated as string delimiters.

Several escape characters are available to you. Another of the most useful ones is the newline character, `\n`, which starts a new line as if by pressing the Enter key. The following code in Example 6-1:

EXAMPLE 6-1 http://actionscriptbible.com/ch6/ex1/

The Newline Character

```
var ok:String = "In an interstellar burst,\nI am back to save the universe."
trace(ok);
```

Will display the following:

```
In an interstellar burst,
I am back to save the universe.
```

Note
Different platforms use different characters for newlines. In most cases, you're safe sticking to \n. But in some cases, you may need to use \r\n for Windows platforms or \r for legacy Mac platforms. ∎

TABLE 6-1

Available Escape Sequences

Escape Sequence	Resulting String
\b	Backspace character.
\f	Form feed character. This character advances one page and is rarely used.
\n	Newline character. Also known as line feed.
\r	Carriage return character.
\t	Tab character.
\u*nnnn*	Inserts a character with the four-digit hexadecimal Unicode code you specify; for example, \u0416 is the Cyrillic character 'zhe' (Ж).
\x*nn*	Inserts a character with the two-digit hexadecimal ASCII code you specify; for example, \x9D is the yen sign character (¥).
\'	Single quote (') character.
\"	Double quote (") character.
\\	Backslash (\) character.

Converting to and from Strings

Because strings are human-readable, they are the preferred method for displaying information about other objects in Flash. For this reason, every object in ActionScript 3.0 inherits the toString() method to represent itself as a String. You can read more about toString() in Chapter 10, "Objects and Dictionaries."

Using toString()

Say you want to display the current date using an instance of the Date class:

```
var now:Date = new Date();
trace("Today's date is " + now.toString());
//Today's date is Mon Jun 15 14:10:20 GMT-0400 2009
```

As you can see, the date is printed in the output as if it were a string. In fact, the `toString()` method is called in many situations automatically. This is a form of implicit conversion, or type coercion, that's built into ActionScript. If you leave out the `toString()` method, as in the following example, the `trace()` method automatically calls it on each parameter passed in.

```
trace(new Date()); //Mon Jun 15 14:10:20 GMT-0400 2009
```

Many classes, including ones you create, do not display much useful information by default when they are converted to strings using `toString()`. The default for most objects is to display the word "object" followed by their class name. For example:

```
trace(new Sprite()); //[object Sprite]
```

Fortunately, you can override this behavior in the classes you create to display whatever useful information you desire. Simply add a custom `toString()` method in your class definition that returns a string. See Chapter 10, "Objects and Dictionaries" for examples.

Casting and Converting to Strings

Now try something else — setting a string to today's date:

```
var now:Date = new Date();
var nowString:String = now; // Causes a compiler error.
```

What happened? Why doesn't this work? Shouldn't `now` be converted to a string implicitly? What's happening is that `nowString` is an object of type `String`; therefore, it is expecting a string object as its value. In cases like this one, the compiler is stricter about what's a `String` and what's not, and you must convert the object to a `String` manually.

```
var nowString:String = String(now);
```

This converts the `Date` object to type `String` and returns `now`'s string equivalent to be assigned to `nowString`. If you need to brush up, check out type conversions and casting in Chapter 4, "Object Oriented Programming."

You could have also used the `toString()` method here:

```
var nowString:String = now.toString();
```

Converting Strings into Other Types

The `String` class can also be converted into some other types fairly painlessly.

Converting Strings to Numbers

Converting a string containing only numerical data is as easy as casting it as a `Number` object:

```
var shoeSize:String = "12";
var iq:Number = Number(shoeSize);
```

Be careful, however, to include numerical characters only. Trying to cast any other value results in a nasty NaN (Not a Number) value being assigned instead:

```
var dialASong:Number = Number("(718) 387-6962");
trace(dialASong); //NaN
```

Note

Although this looks like casting, it's actually calling a global function called Number, which shadows the Number cast operation. Read more about Number conversion in Chapter 7, "Numbers, Math, and Dates." ∎

Adding strings and numbers can be confusing, too, because the compiler converts the numbers to strings rather than the other way around.

```
var a:Number = 2 + "2";
trace(a); //22
```

Assuming that the compiler will do things implicitly for you, even if you're right, is usually a bad idea. Keep your code readable and error free by being explicit about conversions.

```
var a:Number = 2 + parseFloat("2");
trace(a); //4
```

Converting Strings to Arrays

Converting a string to an Array can be useful, especially when processing responses that come from a server that can send only strings. For this, the String class has the split() method. The split() method takes two arguments. The first is the delimiter. This is the character, string, or regular expression that divides the different elements in the array. The second argument is optional and is the maximum number of elements in the new array. It is rarely used. Example 6-2 shows three examples of the split() method in action.

EXAMPLE 6-2 http://actionscriptbible.com/ch6/ex2

The split() method

```
var keywords:String = "people,new york,friends,picnic";
var tags:Array = keywords.split(",");
// tags == ["people", "new york", "friends", "picnic"]

var sentence:String = "The quick brown fox jumped over the lazy dog";
var words:Array = sentence.split(" ", 4); // limit to 4 elements.
// words == ["The", "quick", "brown", "fox"]

var state:String = "Mississippi";
var foo:Array = state.split("ss");
// foo == ["Mi", "i", "ippi"]
```

Combining Strings

To join strings, the `String` class provides the `concat` method, shown in Example 6-3, which takes any number of arguments (converting them to strings if necessary) and returns a new string with all the arguments tacked onto the end of it.

Collected Snippets: The + operator

```
var greeting:String = "Good day.";
var name:String = "Roger";
var s:String = greeting.concat(" My name is ", name, ". How are you?");
trace(s); //Good day. My name is Roger. How are you?
```

However, ActionScript also supports the addition (+) operator for text concatenation, which, you will likely find, is a much more practical way to join strings, as shown in Example 6-3.

```
var s2:String = greeting + " My name is " + name + ". The + operator ROCKS!";
trace(s2); //Good day. My name is Roger. The + operator ROCKS!
```

I've been using the trace statement since the beginning of the book, and it's pretty clear how it works by now. But stop and realize for a moment: you've been doing string manipulation this whole time! Not only have you been converting objects to strings and writing string literals, you've been constructing complex strings out of multiple substrings — all within the parentheses of `trace()`.

In this book, you'll usually see the arguments to `trace()` combined with the + operator as just described. However, you can also pass as many string arguments as you like to `trace()`, and it will concatenate them automatically, separated by spaces.

```
var a:String = "Ground control";
var b:String = "Major Tom";
trace(a, "to", b); //Ground control to Major Tom
```

Converting the Case of a String

To convert a string to the same text in capital letters, simply call the `toUpperCase()` method. As shown in the following example, this returns a new string object with the same letters switched to uppercase. The other nonletter characters are not affected.

```
var cleanser:String = "Ajax";
var webTechnology:String = cleanser.toUpperCase();
trace(webTechnology); //AJAX
```

To switch to all lowercase, use `toLowerCase()` instead:

```
var loudText:String = "CAN YOU HEAR ME?"
var quietText:String = loudText.toLowerCase();
trace(quietText); //can you hear me?
```

Note

Calling these methods does not change the case of the string but returns a new string with the case change. The original string stays the same. If you want to change the original, you have to set its value to the result of the function. For example:

```
var alert:String = "error";
alert = alert.toUpperCase(); // alert == ERROR
```

Using `toUpperCase()` or `toLowerCase()` can be helpful when you need to make sure that two strings are compared without regard to case, as when checking a password:

```
var inputPW:String = "HaXoR"
var storedPW:String = "haxor";
if (storedPW.toLowerCase() == inputPW.toLowerCase()) {
  trace("Login successful");
} else {
  trace("Login error");
}
//Login successful
```

Using the Individual Characters in a String

When you work with strings, you may need to access a specific character in the string or know the total number of characters in the string. You might find it helpful to think of a string as an array of single characters. (C strings are exactly this.) Although a string in ActionScript is not an array, it has some array-like properties. A string has indices starting with 0 and counting up, each containing a single character. You can learn more about arrays in Chapter 8, "Arrays." The following section describes ways to access the specific characters within a string.

Note

ActionScript 3.0 has no special class or type for single characters, like in some other languages. Although I may refer to characters, these are always stored in `String` instances of length 1. ∎

Getting the Number of Characters in a String

To determine the number of characters a string object contains, you can check its `length` property. This property works just like an array's `length` property.

```
var bigWord:String = "PNEUMONOULTRAMICROSCOPICSILICOVOLCANOCONIOSIS";
trace("Number of letters: ", bigWord.length); //45
```

`length` is a read-only property, which means that you cannot change its value explicitly unless you add more characters to your string.

119

Getting a Particular Character

Unlike an array, you cannot access characters in a string using square bracket syntax:

```
var firstName:String = "Adair";
trace(firstName[1]);
```

This produces an error.

To access the individual characters, use the `charAt()` method instead, which returns the character located at the specified index. Character indices, like array indices, are zero-based, so that the character at index 0 in `firstName` is A.

```
trace(firstName.charAt(1)); //d
```

Converting a Character to a Character Code

Similarly, you can convert a character at a specified location into its ASCII character code by using `charCodeAt()`. This may be helpful when you need to work with a character's numerical equivalent instead of the character itself, such as when invoking a Key listener (discussed in Chapter 21, "Interactivity with Mouse and Keyboard Events").

```
var band:String = "Mötley Crüe";
trace(band.charCodeAt(1)); //246 (the character code for ö)
```

This code can be converted back into a letter by calling the static method `String.fromCharCode()`, which returns the converted character as a new `String` object.

```
var buttonPressed:Number = 88;
trace("User pressed ", String.fromCharCode(buttonPressed), " button.");
// Displays: User pressed X button.
```

For a list of ASCII character codes, check out the Wikipedia page `http://en.wikipedia.org/wiki/ASCII`.

ASCII is just one method of encoding textual data. Its major benefit is that every character is exactly 1 byte, which can be a convenient invariant. The tremendous drawback is that it only really works for the most vanilla subset of roman characters; it happens to work perfectly well for English, but almost every other language is out of the question. That's not very friendly, so thankfully you can use another encoding with strings. Read more in the later section "String Encoding and International Text."

Searching within a String

You may need to search within a string for a particular piece of text. For this, the `String` class provides several methods.

Searching by Substring

A substring is any smaller portion of text within a string. For example, "def" is a substring of "abcdefghi". By using the `indexOf()` and `lastIndexOf()` methods, you can search for the first

index of a substring — in other words, the first location of that substring within the text. They operate the same way except that `lastIndexOf()` starts from the end and works backward, whereas `indexOf()` works from the beginning forward. Both methods return the index where the substring begins, or `-1` if the substring is not found. (Remember that string indices are zero based.) Example 6-4 checks the positions of some of the names in a list.

EXAMPLE 6-4 `http://actionscriptbible.com/ch6/ex4`

Collected Snippets: Searching by Substring

```
var names:String = "Jenene, Jake, Jason, Joey, Jaya";
trace(names.indexOf("Jake"));    //8
trace(names.lastIndexOf("J"));   //27
trace(names.indexOf("Robert")); //-1
```

The methods take two parameters each. The first is the substring to search for. The second is an optional parameter specifying the first index to start searching. The second parameter can be useful if you want to find every instance of the substring. The next part of Example 6-4 searches for the letter "a" in the story until it has found all instances.

```
var story:String = "It was a dark and stormy night...";
var pattern:String = "a";
var count:int = 0;
var startIndex:int = 0;
while (story.indexOf(pattern, startIndex) != -1) {
  count++;
  startIndex = story.indexOf(pattern, startIndex) + 1;
}
trace(story);
trace("found " + count + " 'a's"); //Found 4 'a's
```

Searching with Regular Expressions

For even more power, `String` provides methods for searching its text with regular expressions. They include the following:

- `search(pattern:*):int` — Functions similarly to `indexOf` in that it searches the string for any instances of the search criterion. The difference here is that you would use a `RegExp` as an argument; however, `search()` allows you to search for strings or other objects (which are first converted to strings). If nothing is found, it returns `-1`; otherwise, it returns the index of the first match.

- `match(pattern:*):Array` — To return all matches of a particular pattern as an array, use the `match()` method. This also accepts strings or other objects as arguments.

- `replace(pattern:*, replacement:Object):String` — Searches the string for the pattern (usually a `RegExp`) and creates a new string that replaces each match with the replacement object (usually a `String`).

This is only an overview of these methods. Because there is a lot to talk about when it comes to regular expressions, I've given them their own chapter. They are covered in greater detail in Chapter 12, "Regular Expressions."

String Dissection

Sometimes you may need to pull a smaller string out of a larger one. For example, you might want to find the first eight characters of a string, or maybe you only care about the part of a line after a colon. ActionScript offers three methods for doing this: `slice()`, `substr()`, and `substring()`. All these methods are similar in intent. They vary only in the way they handle their two optional parameters.

```
slice(startIndex:Number, endIndex:Number):String
```

and:

```
substring(startIndex:Number, endIndex:Number):String
```

are nearly identical. They both return all the characters between the indexes supplied by the `startIndex` and `endIndex` parameters. The only difference between them is that `slice()` can take negative values as parameters, but `substring()` cannot. Negative values (such as `-3`) count from the end of the string backward.

For both commands, if the end parameter is left out, the end of the string is used by default. If the start is left out, the beginning of the string is used by default.

Example 6-5 searches the string for a space character and uses its index to return the first word.

EXAMPLE 6-5 `http://actionscriptbible.com/ch6/ex5`

Slicing Strings Apart

```
var address:String = "Fourscore and seven years ago...";
var firstSpace:int = address.indexOf(" ");
var firstWord:String = address.slice(0, firstSpace);
trace(firstWord); //Fourscore
var ellipsis:String = address.slice(-3); //takes the last 3 characters.
trace(ellipsis); //...
```

The `substr()` method is slightly different from `slice()` and `substring()`. Instead of taking an end index, this method uses a length parameter. It starts at the supplied start index and returns a string that is as many characters long as the number specified for the `length` parameter. For example:

```
trace(address.substr(4, 5)); //score
```

Because all these methods behave similarly in most situations, you should use whichever one makes the most sense to you. Everything else being equal, I recommend that you use `slice()` rather than `substring()` because it does more and is not as easy to confuse with `substr()`.

String Encoding and International Text

Although it is easy to take for granted, ActionScript's support for multilingual text is actually one of its most compelling features. If you've ever struggled with juggling different text encodings in other languages, you will know exactly what I mean. Because ActionScript's support is thorough, you can use strings in any language without thinking about it. (Displaying them properly can be quite another issue, but more on that in Chapters 17, "Text, Styles, and Fonts," and 18, "Advanced Text Layout.")

The standard that ActionScript 3.0 supports is Unicode. Unicode is a body of standards that's kind of like the UN for computer representations of text. It defines and maintains one character set to rule them all: the ultimate character set for all written languages used on Earth, and some no longer in use, for good measure. It also defines several *encodings*, or ways to encode these characters on binary systems like computers. The worldwide agreement on Unicode is good, and the most common encoding that has emerged is UTF-8. In UTF-8, some characters take more than one byte to represent, but the great thing about it is that the encoding of a given ASCII string is precisely the same in UTF-8 because the mapping of those characters is the same in UTF-8 as in ASCII. Also, those characters are set to take up only one byte in UTF-8 to match ASCII. By the way, this is all extra credit. The important thing to know is this: Flash Player stores strings in UTF-8. It also compiles code in UTF-8, reads files off the internet in UTF-8, writes to disk in UTF-8, and lets you slice and dice strings, as you just learned, in UTF-8. It's a UTF-8 world. The great thing about this is that it *just works*! So your code can contain Unicode characters, your string literals can, and you can operate on them correctly, as shown in Example 6-6.

EXAMPLE 6-6 `http://actionscriptbible.com/ch6/ex6`

Unicode Characters

```
var jp:String = "平和";
var he:String = "שלום";
var ar:String = "سلام";
```

You can have Unicode characters in comments as well. However, stick to ASCII for your actual code, even if your native written language isn't English or in the Roman alphabet.

You have to be careful of encodings when dealing with technologies that don't support UTF-8, or for those that do but must be told specifically to expect it. Most network services fall under one of these categories. For truly deep control over encodings, you can write strings into and read strings out of binary data using `ByteArray`, optionally converting to one of the dozens of supported encodings along the way. Read more about binary data in Chapter 13, "Binary Data and Byte Arrays."

Summary

- A string can be any piece of text. Quotation marks identify strings from code that should be executed.
- The `String` class provides loads of additional functionality. All strings are instances of the `String` class even if they are string literals.

- Creating a custom `toString()` method in your classes can help you understand the data contained within them when logging or using traces.

- Strings can be cut up and combined, and they have individual characters accessed much like arrays.

- Strings in ActionScript are case sensitive, but converting cases is easy using the `toUpperCase()` and `toLowerCase()` methods.

- You can search within strings with simple methods like `indexOf()`, or you can use regular expressions to match complex criteria.

- Strings are stored in UTF-8, and any international text and symbols defined in Unicode may be used.

Numbers, Math, and Dates

Without good support for numbers, there's not much you can program. It should be no surprise, then, that ActionScript 3.0 enables all kinds of numeric activity, from basic arithmetic to date and time handling. With the information in this chapter, you can start making useful, interactive, and aesthetic computations.

Understanding Numeric Types

Using ActionScript, you can create numbers of different types. To understand how to use these, however, you must take a step back and look at the sets of numbers themselves and what they represent. Later, you will review how different kinds of numbers are represented on modern computers. Knowing the implications of these implementation details can help the wary programmer understand which data types are appropriate in which situations, the limitations of the language, and common problems that might arise. If you're familiar with numeric types in other languages, you might want to skip ahead to "Using Numbers in ActionScript."

Sets of Numbers

A number is a number is a number, right? Not so; there are several types of numbers. These types of numbers are defined by the set of values that they can represent.

Most familiar might be the set of natural numbers, or $\mathbb{N}$. This set contains whole numbers starting with zero: $\{0, 1, 2, 3, \ldots\}$. These are the numbers you use to count discrete objects: "I have two parents." "There are four emergency exits on this aircraft." "This piano has 88 keys."

A superset of the natural numbers is the set of integers, represented by $\mathbb{Z}$. This set contains whole numbers both negative and positive: $\{ \ldots, -2, -1, 0, 1, 2, \ldots \}$.

An even larger set is the set of real numbers, represented by $\mathbb{R}$. Real numbers include both rational numbers and irrational numbers — that is, numbers that can be represented by a fraction and those that can't. It's impossible to exhaustively list any subset of them because between any two real numbers lie an infinite number of other real numbers. Some examples, however, are -10, $6\frac{1}{2}$, 3.14159265 ..., and 4.4. (Of these examples, the number π, approximated by 3.14159265..., is irrational. π, read "pi," can't be accurately represented with a fraction or decimal expansion.) Any number you can write with a fraction or a decimal is a real number. You can measure things such as distances, angles, pH, and pounds of cookies with real numbers.

Representing Numbers

When you write down numbers, you are representing their values with digits. For instance, the concept of one hundred is a one followed by two zeros. But you could also express this as one hundred tick marks, $1 * 10^2$, the Roman numeral *C*, or with ten rows of ten ones. With some imagination, you could come up with lots of different ways to represent a single number.

The common way we write numbers is in base 10, also known as decimal. In our system of writing, there are ten basic digits, 0 through 9, and every place for a digit represents a factor of ten more than the place to its right. Thus, 342 is three hundreds plus four tens plus two ones.

When programming applications, you may find yourself using other bases, most likely base 16, also known as hexadecimal. Base 16 has 16 basic digits, 0 through 9, followed by A, B, C, D, E, and F. A digit in each place represents 16 times the value of the digit to its right. The number written 12 in hexadecimal is one sixteen plus two ones: eighteen. But because the 0–9 digits look the same in both systems, there's no way to know whether a number written 12 is meant to be in hexadecimal, where it means sixteen, or decimal, where it represents twelve. To differentiate between hexadecimal and decimal notations, we typically precede hexadecimal numbers with 0x. The number 0x2A is two sixteens plus A (ten) ones, or 42 in decimal.

Why use hexadecimal? If you have tried talking to a computer, you know that computers don't understand decimal numbers. They understand only base 2, or binary, where the only digits are 0 and 1, and each place represents twice the value of the digit to the right of it. However, it takes a lot more space and effort to write in binary — for humans like us, that is. For example, 2009 is expressed in binary as 11111011001. Hexadecimal is both compact and nicely compatible with binary. Each digit in base 16 is represented by four digits in base 2, because 16 is 2^4. So 2009 becomes 0x7D9, which is much more compact despite the space taken up by 0x. Hexadecimal is used in ActionScript 3.0 most frequently to write 32-bit color values, as you'll see later in the chapter.

Digital Representations of Numbers

Representing a number in a computer presents a unique set of challenges. First, computers represent everything as binary. Second, the way your CPU works sets limits on the number of binary digits that can be stored as a single value.

Unsigned Integers

The simplest kind of number to represent digitally is a natural number. Natural numbers (positive whole numbers) are represented by *unsigned integers* on a computer.

The integer type in ActionScript 3.0 is 32 bits. This means that there are 32 places for binary digits. To understand the limit this imposes, consider a comparison to decimal numbers. The highest number you can write in base 10 with 3 digits is 999, which is one less than 1,000 or 10^3. So the largest number you can write in base 2 with 32 digits is

    1111 1111 1111 1111 1111 1111 1111 1111

which is one less than

```
1 0000 0000 0000 0000 0000 0000 0000 0000
```

or 2^{32}. This means that the biggest number you can represent with a 32-bit unsigned integer is $2^{32} - 1$, which works out to 4,294,967,295 in base 10.

Four billion is a large number, to be sure, but it's not hard to imagine needing a bigger number, like the population of the world, or the number of hits some web sites get in a year. It is important to remember that numbers represented on your computer are limited. As I've just demonstrated, the maximum value of unsigned integers is limited directly by the size of the unsigned integer in memory — in ActionScript 3.0, it's 32 bits.

Signed Integers

Integer numbers (whole numbers that might be either positive or negative) are represented by *signed integers* on a computer. The term "signed" refers to the fact that these numbers carry information about their sign — that is, whether they are positive or negative. Unsigned numbers, in contrast, do not carry information about their sign. They do not say whether they are positive or negative. It is only through convention that we associate them with positive numbers, just as we expect maps to represent north as up.

But if all you have at your disposal to represent both positive and negative numbers are 32 binary digits, how do you represent both positive and negative numbers? The solution is both simple and clever. There are two signs: positive and negative. There are two possible values for one binary digit, 0 and 1. So if you just reserve one of the 32 bits you have at your disposal for the sign, you can have 31 bits left to store the value of the number.

This trade-off affects the maximum absolute value that can be stored. If we reserve 1 bit for the sign, a signed integer will have 31 bits left to represent its absolute value. A binary number with 31 digits can only go up to $2^{31} - 1$, or about 2 billion. By taking one binary digit away, you halve the maximum value. But wait! By adding a sign, you double the number of values that can be represented! Every number that was once an unsigned value (except zero) can now be two values: positive or negative. Signed integers, instead of going as high as unsigned integers, go half as high ... but equally as low. The minimum value of a 32-bit unsigned integer is 0, but the minimum value of a 32-bit signed integer is −2,147,483,648.

The sign bit is stored as the most significant bit of the 32. But in addition to this, conventional languages such as ActionScript 3.0 use a trick to let the CPU add two numbers the same way regardless of their sign. This trick is called two's complement, and it involves counting backward in binary for the nonsign digits of negative numbers. For instance, a 4-bit signed integer for −1 would be 1111, −2 would be 1110, and so on. Then, adding a positive and a negative number is a normal binary addition, with the overflow ignored. I mention this because overflows in integer arithmetic are ignored, so if you are not careful with boundary cases, you can attain deceptively incorrect results as a result of an overflow. You must also be aware of this when using bit math with signed integers.

Floating-Point Numbers

Real numbers are represented by *floating-point* numbers on a computer. Just as scientific notation can represent a huge range of numbers concisely in the form a $* 10^b$, floating-point numbers have an incredible range, and their accuracy is proportionate to the scale of the value itself. For example, 5.12 $* 10^{-5}$ is an incredibly small and precise number, 0.0000512, and 5.12 $* 10^9$ is a very large but

imprecise number, 5,120,000,000. However, both express sufficient precision for their own scale. You don't care about an error of a thousandth of a gram when measuring the weight of a planet, but you need that accuracy when measuring chemical compositions.

Floating-point numbers break up the available bits to store a sign, a significand, and an exponent. The significand in a floating-point number is its fractional part — the part that encodes the number to be scaled up or down, or a in the expression $a * 2^b$. The exponent is the scale of the number, or b in the expression $a * 2^b$. ActionScript 3.0 uses a standardized implementation of floating-point numbers defined by the IEEE Standard for Binary Floating-Point Arithmetic, or IEEE 754. In this implementation, both a and b are binary, and the base itself is 2.

The floating-point type in AS3, introduced in the material that follows, is a 64-bit double-precision IEEE 754 floating-point number. It reserves 52 bits for the fractional part and 11 bits for the exponent, with 1 bit left over for the sign. This size structure can represent incredibly small (in absolute value) and large numbers with great accuracy. It's impossible to represent all real numbers with perfect accuracy in a computer, so floating-point numbers often approximate rather than represent values, but the accuracy afforded by double-precision floating-point numbers should be sufficient for almost any task. Use floating-point numbers to represent real numbers and extremely large positive or negative numbers. The maximum value AS3 can represent with a double-precision floating-point number is about $1.8 * 10^{308}$. It's incredibly difficult to give a number that big any perspective. For one measure, it's roughly 1,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000,000 times as many atoms as exist in the visible universe.

Using Numbers in ActionScript

ActionScript 3.0 includes three base types for numbers, which reflect the three ways I covered to represent numbers digitally, allowing the programmer to choose the best number type for the task at hand. The number types include special values to keep your program from breaking in certain edge cases. Additionally, the language allows you to type numbers directly into code.

Number

The `Number` type is an IEEE 754 double-precision 64-bit floating-point number and, as such, it can represent positive and negative numbers, both whole and fractional. `Number`s are applicable for almost any kind of measurement and are one of the principal workhorses of ActionScript 3.0.

The default value for an unassigned variable of type `Number` is `NaN`, which is defined in the section "Edge Cases" later in this chapter. Usually `Number`s are assigned by setting them equal to a numeric literal, but the `Number` class also has a constructor, which attempts to convert the object passed in to a number and assigns it to the new `Number`. Typical usage of a `Number` is as follows:

```
var n:Number;
trace(n); // NaN
n = 12;
trace(n); // 12
```

Tip

Assignment is the typical way to associate a value with a variable; it is rare to see a numeric type's constructor in use. ■

Use `Numbers` for any kind of measurement, for values that might contain fractional parts, and for extremely large and small values.

int

The `int` type is a signed 32-bit integer. Despite its lowercase name, it is, like all types, a class. Like `Number`, `int` has a constructor that is rarely used. You can create a new `int` by merely typing a number literal. If you try to assign an `int` variable a nonwhole number, the fractional part is dropped (rather than rounded). The default value for an unassigned `int` value is 0.

Integer values in Flash Player take up less memory than floating-point numbers, so you can use integer number types for sections of code that need to be maximally optimized, and where fractional parts do not matter. The `int` type is perfect for counters and is frequently found in `for` loops:

```
for (var i:int = 0; i < 1000; i++)
```

uint

Another new numeric type found in ActionScript 3.0, `uint` is an unsigned 32-bit integer. It, too, is a class and has a rarely used constructor. Like `int`, its default value is 0. In fact, `uint` is identical to `int` except in its range of values.

The `uint` type is used for bit math because its value is always a simple binary translation of the value, without special encoding or rules like counting backward in two's complement. It provides unfettered access to all 32 bits of the value in memory. A good example of this is color values, illustrated in Figure 7-1, which pack information for transparency and amounts of red, green, and blue into a single 32-bit field.

A color value represented in a single uint

```
0 x F F  A 0  A 0  A 0
   Alpha  Red  Green  Blue
```

The value in Figure 7-1 represents a solid gray. Because it uses all 32 bits, it can't be stored in a signed integer, which uses 1 bit for sign and leaves 31 bits for use. You'll see much more of `uint` in Chapter 13, "Binary Data and ByteArrays."

Caution

You should not use the `uint` **type for integer numbers or counters because it is unable to represent negative numbers. Even if you know the integer you want to represent won't go under zero, it's a good idea to leave it signed. Otherwise, it's easy to make some simple errors. Use** `uints` **for bit math and** `ints` **for all other integers.**

Furthermore, it's good to remember that integers behave badly when they overrun their limits, wrapping over into the opposite sign for signed integers and between zero and the maximum value for unsigned integers, whereas `Numbers` **simply become less accurate as they increase radically. Don't use** `ints` **or** `uints` **for values that have even a remote chance of exceeding their limits, or be sure to check these values. In general, use** `Number` **for all numeric types, because it is the most flexible. ∎**

Literals

Using number literals is simple. The easiest way to enter a number directly in code is to write it in normal decimal notation. Valid decimal literals include

```
1337;
-4;
.8;
0.333;
-1.414;
```

As I briefly mentioned, you denote hexadecimal numbers by preceding them with 0x. To interpret a number as hexadecimal in ActionScript code, you do the same thing:

```
var foo:uint = 0x12AB;
```

Note

When typing A–F **hexadecimal characters, case is ignored.** `0xA` **is equivalent to** `0xa`**. ∎**

You can also use exponential notation to declare numbers. Exponential notation expresses a number as a real number times a power of ten. Typically this is seen in scientific notation, where the real number part is always between 1 and 10. This kind of representation lets you focus on the relative sizes of numbers without expressing them in a lengthy string of digits, especially for very small or very large numbers, such as $6.02 * 10^{23}$. When writing exponential notation in code, you use the character e to represent the base and to indicate that the value following it is the exponent. This same number, then, would be written as `6.02e23`. The following are all valid exponential literals:

```
2.007e3; // 2007
1414e-3; // 1.414
1.9e+17; // 190,000,000,000,000,000
```

Caution

In earlier versions of ActionScript, you could also enter literal numbers in base eight, or octal, by preceding them with an unnecessary 0. This feature is likely to have caused more inadvertent errors than triumphs for octal notation, and it has been removed from ActionScript 3.0. ∎

Edge Cases

Each ActionScript type includes special values that are included in its possible range. These are indispensable for preventing your program from producing errors when certain edge cases occur.

Not a Number

The constant NaN represents "not a number," a nonnumeric value included in the Number type. NaN is found in instances of Number that have not been assigned a value, as the result of failed conversions. Mathematical operations with nonreal or undefined results also yield NaN, such as the square root of −1 or 0 divided by 0.

Comparisons with NaN always return false, and most mathematical operations on NaN result in NaN. To check whether a number is defined, don't try to compare its value to NaN. Instead, use the top-level function isNaN(), as follows:

```
var n:Number;
trace(n); //NaN
trace(isNaN(n)); //true
trace(n == NaN); //false! That's why you use isNaN()
n = Number("this won't convert into a number");
trace(isNaN(n)); //true
n = 10;
trace(isNaN(n)); //false
```

In ActionScript 3.0, Number instances can be only NaN or a real number — never undefined or void.

Infinity

The Number type also has special constant values for positive and negative infinity: Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY. If you end up with an infinite value, for instance by dividing a Number by zero, instead of an overflow or runtime error, the Number takes on one of the special infinite values. You can check infinite conditions through comparison, and the comparisons work as you might expect. Any noninfinite number is less than positive infinity and greater than negative infinity, although infinity equals itself.

```
var n:Number = 10 / 0;
trace(n); //Infinity
trace(n == Number.POSITIVE_INFINITY); //true
trace(isFinite(n)); //false
```

Minimum and Maximum Values

Not counting infinity, physical limits are imposed on the size of numbers based on their implementation in the ActionScript Virtual Machine. For instance, you learned that 32-bit unsigned integers can go up to only $2^{31} - 1$. These real limits are documented by the MAX_VALUE and MIN_VALUE static constants of all three number classes. The constants refer to the overall maximum and minimum possible values for int and uint. For Number, they refer to the largest positive finite number that can be represented and the smallest nonzero, nonnegative number that can be represented:

```
trace(uint.MIN_VALUE); //0
trace(uint.MAX_VALUE); //4294967295
trace(int.MIN_VALUE); //-2147483648
trace(int.MAX_VALUE); //2147483647
trace(Number.MIN_VALUE); //4.9406564584124654e-324
trace(Number.MAX_VALUE); //1.79769313486231e+308
```

NaN and Infinity are concepts that apply to the Number type only. Integers, signed and unsigned, don't have these fail-safes.

Manipulating Numbers

Numbers of all kinds in ActionScript 3.0 are flexible and can serve whatever purpose you need them to. If this includes some type-bending, all the better. Casting and conversion are easy in AS3.

Numeric Conversions

If you want to multiply two values, a floating-point number and an integer, when what you really need is an integer, don't sweat it. ActionScript 3.0 automatically performs necessary conversions between floating-point and integer numbers in expressions based on the left side of the expression: what type it will be assigned to or returned as. If it's not declared which type the expression should be evaluated as, expressions are upgraded to floating-point when any part of them results in a floating-point component:

```
var i:int = 3 * 2.04; //adding an int and a float, assigning to an int
trace(i); //6 (it's an int!)

var n:Number = i + 0.1; //adding an int and a float, assigning to a float
trace(n); //6.1 (it's a float!)

var x = 2 / 3;
//dividing an int and an int, upgraded to float since the result is
//not a whole number, and the left side doesn't specify a type
trace(x); //0.6666666666666666 (it's a float!)
```

String Conversions

Probably the most useful kind of conversions for numbers is to and from `String`s. You can include numbers in messages, format them, and read them in from user input. Thankfully, this process is as simple as a cast, and in many cases it can be handled automatically.

To convert a number to a string, you have a few options:

- Call the number's `toString()` method. This is the preferred way.

- Explicitly cast the number to a `String`. This actually calls the top-level `String()` conversion function, which interprets the number as a string.

- Include the number variable in a string expression. This implicitly coerces the number to a `String`.

  ```
  for (var i:int = 99; i > 0; i--) {
    trace(i + " bottles of beer on the wall");
  }
  ```

- Call one of the special formatting methods on the number variable to print it to a string with more control. All three number types have methods `toExponential()`, `toFixed()`, and `toPrecision()`, for formatting the number in exponential and fixed-point notation and limiting the number of decimal places printed out.

Getting a number back from a string value is as easy as explicitly casting the string to a number type. To parse a number out of a string that might have other text in it, ignoring the other text, you can use the top-level `parseInt()` and `parseFloat()` functions to return integer and real number interpretations of the numeric content of a string. Also, `parseInt()` and `parseFloat()` allow you to

interpret the text as a number in an arbitrary base. The following snippet compares approaches to converting strings to numbers:

```
trace(parseInt("3.14")); // 3
trace(int("3.14")); // 3
trace(parseFloat("3.14")); // 3.14
trace(Number("3.14")); // 3.14
trace(parseFloat("3.14 has a posse")); // 3.14
trace(Number("3.14 has a posse")); // NaN
trace(parseFloat("What's up 3.14?")); // NaN
```

The last line doesn't work because `parseInt()` and `parseFloat()` require the number to be the first thing in the string, which makes these functions a lot less useful.

Of these two approaches, I recommend using the explicit cast method. Because you really are changing types, a cast in your code looks better than calling a global function like `parseInt()`.

Cross-Reference

You can use regular expressions to easily parse out numbers from complex strings and do a much better job than the `parseInt()` and `parseFloat()` functions. See Chapter 12, "Regular Expressions," to learn about applying regular expressions. ∎

Performing Arithmetic

ActionScript 3.0 supports all the basic arithmetic you'd find on any calculator, and expressions are written out as you might write them on paper. AS3 follows the correct order of operations, so

```
1 + 2 * 3
```

returns 7 as it should, instead of 9 if it simply operated left to right. Arithmetic and order of operations were introduced in Chapter 2, "ActionScript 3.0 Language Basics." Table 7-1 summarizes the basic operators for arithmetic in AS3.

TABLE 7-1

Basic Arithmetic Operators

Operator	Meaning
`a + b`	The sum of `a` and `b`.
`a * b`	The product of `a` and `b`.
`a - b`	`a` minus `b`.
`a / b`	`a` divided by `b`.
`a % b`	`a` modulo `b` (the remainder of `a/b`).
`-a`	The negative of `a` (-1 times a).

continued

133

TABLE 7-1 *(continued)*	
Operator	**Meaning**
(a + b)	Evaluate subexpressions in parentheses first.
a++	Add 1 to a after evaluating the expression.
++a	Add 1 to a before evaluating the rest of the expression.
a--	Subtract 1 from a after evaluating the expression.
--a	Subtract 1 from a before evaluating the rest of the expression.

Combining operators with assignment uses the left side of the expression as the first operand. For example,

```
a = a + 10;
```

can be more concisely written as

```
a += 10;
```

Beyond these simple operators, there are static methods in the Math utility class, as shown in Table 7-2, that enable you to perform more arithmetic.

TABLE 7-2

Math Class Arithmetic

Method Call	**Returns**
Math.pow(a, b)	a raised to the b power (a^b)
Math.exp(a)	e raised to the a power (e^a)
Math.floor(a)	a rounded down
Math.ceil(a)	a rounded up
Math.round(a)	a rounded to the nearest digit
Math.max(a, b, c...)	Maximum of the set a, b, c . . .
Math.min(a, b, c...)	Minimum of the set a, b, c . . .
Math.sqrt(a)	Square root of a
Math.abs(a)	Absolute value of a
Math.log(a)	Logarithm (base 10) of a
Math.ln(a)	Natural logarithm (base e) of a

Between built-in operators and the methods of the Math class, ActionScript 3.0 gives you a solid basis with which to make computations.

Performing Trigonometric Calculations

Also built into ActionScript 3.0 are trigonometric functions. Flash is so often employed to create interactive graphics, and these can rarely be done without a helping of trigonometry. Therefore, the `Math` utility class also includes the methods shown in Table 7-3.

TABLE 7-3

Math Class Trigonometry

Method Call	Returns
`Math.sin(a)`	Sine of an angle measuring `a` radians
`Math.cos(a)`	Cosine of an angle measuring `a` radians
`Math.tan(a)`	Tangent of an angle measuring `a` radians
`Math.asin(a)`	Angle in radians whose sine is `a` (arcsine of `a`)
`Math.acos(a)`	Angle in radians whose cosine is `a` (arccosine of `a`)
`Math.atan(a)`	Angle in radians whose tangent is `a` (arctangent of `a`)
`Math.atan2(y, x)`	Angle which, drawn from the origin, intersects the point (x, y) (arctangent of y/x)

In addition, the `Math` class includes the constant `Math.PI` for the number pi, ratio of a circle's circumference to its diameter.

Tip

All trig functions operate on radians, an angular unit in which 2π radians measure a full revolution. All display objects' rotations are measured in degrees, in which 360 degrees measures a full revolution. You can use the equality π radians = 180 degrees to translate between them easily:

```
valInRadians = valInDegrees / 180 * Math.PI;
valInDegrees = valInRadians / Math.PI * 180;
```

In Example 7-1, you'll create a game-like scenario where a turret points toward the crosshairs that track your mouse movements. This is made possible by the arctangent function, which finds the angle of a corner of a right triangle given the ratio of the lengths of the edges that form the angle. In a game like this, when you want some object A to orient itself toward another object B, the position of object B relative to A defines the hypotenuse of the triangle. This is shown in Figure 7-2.

EXAMPLE 7-1 `http://actionscriptbible.com/ch7/ex1`

Trigonometric Functions

```
package {
  import flash.display.*;
  import flash.events.Event;
```
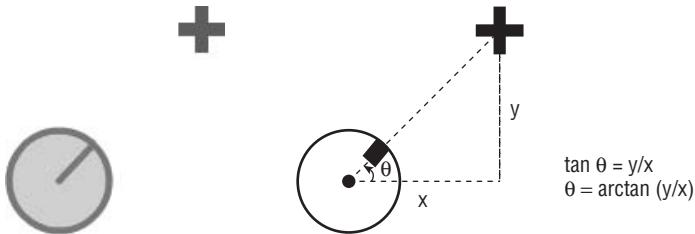
EXAMPLE 7-1   *(continued)*

```
import flash.ui.Mouse;

public class ch7ex1 extends Sprite {
  protected var compass:Sprite;
  protected var crosshairs:Sprite;
  public function ch7ex1() {
    drawCompass();
    drawCrosshairs();
    addEventListener(Event.ENTER_FRAME, tick);
  }
  protected function tick(event:Event):void {
    Mouse.hide();
    crosshairs.x = stage.mouseX;
    crosshairs.y = stage.mouseY;
    compass.rotation = Math.atan2(
      stage.mouseY - compass.y,
      stage.mouseX - compass.x)
      * 180 / Math.PI;
  }
  protected function drawCompass():void {
    var RADIUS:Number = 30;
    compass = new Sprite();
    var g:Graphics = compass.graphics;
    g.lineStyle(4, 0xe03030);
    g.beginFill(0xd0d0d0);
    g.drawCircle(0, 0, RADIUS);
    g.endFill();
    g.moveTo(0, 0);
    g.lineTo(RADIUS, 0);
    addChild(compass);
    compass.x = stage.stageWidth / 2;
    compass.y = stage.stageHeight / 2;
  }
  protected function drawCrosshairs():void {
    var SIZE:Number = 10;
    crosshairs = new Sprite();
    var g:Graphics = crosshairs.graphics;
    g.lineStyle(8, 0x4040f0, 1, false, LineScaleMode.NONE, CapsStyle.SQUARE);
    g.moveTo(0, SIZE);
    g.lineTo(0, -SIZE);
    g.moveTo(SIZE, 0);
    g.lineTo(-SIZE, 0);
    crosshairs.blendMode = BlendMode.MULTIPLY;
    addChild(crosshairs);
  }
 }
}
```

FIGURE 7-2

Output of Example 7-2. Using trigonometry to implement a turret aiming system.

$$\tan \theta = y/x$$
$$\theta = \arctan(y/x)$$

Generating Randomness

Incorporating elements of randomness into your program is an excellent way to simulate lifelike, unpredictable behavior. This can be applied to animate graphics for living creatures or to create generative art, often producing real beauty from a chance arrangement. Subtly altering the appearance of your program with randomness is a common technique for evoking an organic, always-new feeling. You can apply randomness to any quantifiable property, and in whatever degree you desire.

Given the many applications of chaos, generating some in ActionScript 3.0 is trivial.

```
Math.random();
```

This method generates a pseudorandom number between 0 and 1. It could potentially be 0, but it will never get as high as exactly 1 ($0 \leq n < 1$). In other languages, you may be required to set the seed value for the random number generator, but in ActionScript 3.0 this is not available or necessary.

Note
Pseudorandom numbers are determined by a series of repeatable steps that a computer can execute. This means that they are not truly random. However, they have a random distribution and for almost every application are sufficient. For cryptographic applications you should consider writing your own random number routines and incorporating more chaotic variables into them, such as sampling mouse movement or background noise from a microphone. ∎

The `Math.random()` method generates unit-scale random numbers only. More than likely, you need the values to lie in a different range than 0 to 1. Remember that the `Math.random()` function does not take arguments; you should scale the numbers to the desired range yourself. A simple formula for producing a random number between two values is presented here:

```
function randomInRange(min:Number, max:Number):Number {
    var scale:Number = max - min;
    return Math.random() * scale + min;
}
```

The fact that `Math.random()` returns a number between 0 and 1 makes it easy to manipulate. Here, you simply multiply it by the size of the range and shift it up by the minimum value.

Manipulating Dates and Times

When you create applications that interface with the world at large, timing becomes a huge concern. As individuals, we want to mark off the years since we were born, be on time for flights, plan a party for a specific time and day, know important events in our past and our history, and mark the passage of time. Time is a linear variable; we all experience it at the same rate forever. (Actually, we don't, but I'll save that for *Physics Bible*.) Therefore, we can represent time as a single dimension, measuring it with ever-increasing numbers. I could say, "I let my tea brew for 180 seconds," and you would understand that measurement of time. So, too, could any computer or simple pocket calculator because a number is just a number.

However, things get really complicated when you try to manipulate and communicate dates. Most of us use a complex and fairly hacked-up system to break up time into cycles that roughly sync up with earth's travel around the sun, its rotation, the location of the moon, the seasons, and all kinds of events. This mess of rules and exceptions is called the Gregorian calendar. Times during the day have problems, too, as everyone on the globe wants noon to be about when the sun is in the middle of the sky; we're left with the same instant being represented as different times across the world: time zones. All of these we have become somewhat adept at manipulating mentally. I could tell you what month it was 50 days ago, but without a lot of thought, I couldn't tell you what day of the week it was. For both humans and computers, calculating dates requires knowledge of how our systems of measurement work.

ActionScript 3.0 knows all these rules and lets you manipulate times and dates in their common units, rather than simply as a raw number. There is one class in ActionScript 3.0 that represents a single moment in time, including both time and date information. This is the Date class.

Creating a Date

Let's start using Dates by creating one that represents "right now," or more precisely, "the instant in which the Date object was created." The default use of the Date constructor will do just this.

```
var now:Date = new Date();
```

The Date constructor is one of the more versatile constructors in ActionScript 3.0. With it, you can construct not just Dates for right now, but Dates representing any point in time you pass in.

You can send the Date constructor a series of values for the year, month, day of the month, hour, minute, second, and millisecond to represent. You must include at minimum two of these parameters — at least the year and month — and all others will default to zero (the stroke of midnight the first day of the month), as shown in Example 7-2.

EXAMPLE 7-2 http://actionscriptbible.com/ch7/ex2

Collected Snippets: Using Date

```
var bday:Date = new Date(1981, 4, 12); // May 12, 1981
var timeOfWriting:Date = new Date(2009, 5, 29, 22, 25);
// Mon Jun 29 22:25:00 GMT-0400 2009
```

Hours are measured in 24-hour/military time, so they can range from 0 (12 a.m.) to 23 (11 p.m.).

You might notice that there's something fishy with the dates as entered in the preceding code. Certain units in the `Date` class are zero-indexed, and certain units are one-indexed. Although January is the first month, the `Date` class calls it zero. Properties are zero-indexed when they refer to named units, such as months (January, February, and so on) or days of the week (Monday, Tuesday, and so on). This way, you can use an array of month names to generate human-readable dates, as shown in Example 7-2.

```
var daysOfWeek:Array = ["domingo", "lunes", "martes", "miércoles",
                        "jueves", "viernes", "sábado"]
trace(daysOfWeek[timeOfWriting.day]); //lunes
```

By representing these named parts of time as numbers, they remain language independent.

Months and days of the week are indexed from zero. Days of the month are indexed from one. Hours, minutes, seconds, and milliseconds are measured from zero. Years are measured from zero (AD 0/CE 0).

A third way to create a `Date` object is by passing it a string representing a time. The string must contain at minimum a year, month, and day of the month, but it can contain more. If the `Date` constructor can't figure out how to get a date out of your string, it creates a new `Date` with an `Invalid Date` value, the equivalent of a `NaN`. All of the following lines create valid dates:

```
new Date("1/28/2007"); // United States style MM/DD/YYYY
new Date("Sun Jan 28 2007");
new Date("28 Jan 2007");
new Date("Sun Jan 28 2007 3:16:12 PM");
```

The parsing function is only somewhat lenient; it is better suited for interpreting specifically computer-formatted dates than free-form human input. It's always a good idea to check a date's validity after constructing it from a string. You can check a date's validity by checking whether its epoch time is a number.

```
var invalid:Date = new Date("Yesterday");
trace(invalid); // Invalid Date
trace(isNaN(invalid.valueOf())); // true
```

So what is this so-called epoch time? I don't know. Hmm... maybe somewhere in this big book there is a section called ...

Epoch Time

In simpler times, people didn't have `Date` classes or fancy laptops or terabyte hard drives full of *30 Rock* episodes in HD. Folks then, they had a big ol' heap of memory and some simple types that stored some simple values. When they needed a time value — and people always needed a time value — they relied on their old friend `int`. Remember that old tale about time just being a number? Well, it sure can be, if you can just decide what time was 0. So those folks did just that. They stored a time as a number of seconds since another time, and they put that number right in an `int`.

What was the time those folks measured against? It was the Epoch: midnight, January 1, 1970. But why that day? The epoch of what? This is the way time has always been represented in Unix, and

that day was the birthday of Unix (or so they say). In fact, this kind of time measurement is usually referred to as Unix time, but because it is applied so widely now, I feel it's more diplomatic to call it epoch time.

Of course, by putting that number in a signed `int`, they created a problem. Can you guess what it is? A 32-bit signed integer can represent only a time up to 2^{31} seconds before or after the epoch. That means dates before December 1901 and dates after January 2038 are out of the range of epoch time when stored in an `int`. In those simpler times, 2038 must have looked like a long way away.

Thankfully, ActionScript 3.0 uses a `Number` type to represent epoch time. This means that you'd have to be out of your mind to come up with a date it can't represent (remember `Number.MAX_VALUE?`), and it also gives you the precision to measure epoch time as *milliseconds* since the epoch.

Caution

Epoch time in ActionScript 3.0 is represented in milliseconds since January 1, 1970, not seconds. ■

In the present, epoch time is still a useful tool. Because it's just a number, you can play with it freely, using arithmetic instead of `Date` methods when it's more convenient. To selectively use epoch time, you have to be able to convert a `Date` object to an epoch time and back; methods covered in the following section, "Accessing and Modifying a Date," show how to do that. In addition to those methods that fit in with other means of modifying a `Date`, you can pass epoch time to the `Date` constructor to create a new `Date` instance from an epoch time, as shown in Example 7-3.

EXAMPLE 7-3 `http://actionscriptbible.com/ch7/ex3`

Creating a Date from an Epoch Time

```
var d:Date = new Date(0);
trace(d); // Wed Dec 31 16:00:00 GMT-0800 1969
```

In addition, if you want nothing to do with `Date` objects and just want to use epoch time, you can parse strings into dates without creating a new `Date` object by using the `Date` class's static `parse()` method:

```
trace(Date.parse("1/1/1970")); // 28800000
```

Did you notice something wrong with these examples? Get a different value when you executed them? Shouldn't the first one return January 1, 1970 and the second one 0? They would, if you lived on the Greenwich meridian.

Time zones

As you know, one of the complications of our way of measuring time is the separation of our globe into time zones, where the same instant is a different time at different longitudes. The `Date` class also takes this into account.

When you create a new `Date` object, ActionScript assumes you want it in your local time zone unless you specify a time zone, as in `new Date("Jan 28 19:46:00 GMT-0500 2007")`. Once you have a `Date` object, you can operate on it and reference it in either your local time zone or UTC.

UTC, or Coordinated Universal Time, is the same thing as GMT, Greenwich Mean Time, and is the "neutral" time zone. So if you want two people on different parts of the globe to agree on what to call a certain time, they can both reference it by the same time zone to call it the same time. That time zone is UTC or GMT, the time zone over the Greenwich meridian in Great Britain. So by referencing times with respect to UTC, you can avoid geographical implications.

Because epoch time is measured relative to a specific instant, you need that reference point to be fixed. The epoch is not just January 1, 1970 at midnight; it's January 1, 1970, midnight at UTC. Therefore, if I am in Los Angeles using Pacific Time, GMT-0800, the *local* time of the epoch is at plus eight hours, or 28,800,000 milliseconds like the example showed.

In the first part of the example, you created a new Date object at the actual epoch time, but you printed it in our local time. For Californians, that's December 31, 1969 at 4 p.m. When it's midnight in UTC, it's 4 p.m. the previous day in Pacific time. Your results may vary as you run the example from your local time zone. Even my results completely changed, as I revised this chapter in Brooklyn during Daylight Savings Time (GMT-0400).

In the second part of the example, you created a new Date to represent January 1, 1970 in our local time. Epoch time is measured from January 1, 1970 UTC, so again in California the difference is eight hours, or 1,000 ms/sec * 60 sec/min * 60 min/hr * 8 hr = 28,800,000 ms. Try running the example yourself and verifying your own time zone.

Accessing and Modifying a Date

Once you have a date stored in a Date variable, you can work with it in the natural units of time thanks to accessor methods and functions of the Date class. You can set or read any unit of time, in either UTC or in the local time zone. It's your choice whether to use the explicit accessors (methods) or implicit accessors (properties) to read and write these values.

For each property of a date and time listed in Table 7-4, the accessors are named as follows:

- get[*Property*]() — Read the property from the Date object in local time.
- getUTC[*Property*]() — Read the property from the Date object in UTC.
- set[*Property*]() — Set the property in local time.
- setUTC[*Property*]() — Set the property in UTC.
- [*property*] — Read or write the property in local time.
- [*property*]UTC — Read or write the property in UTC.

So, for example, you can access the hour of a date with any of the following:

```
date.hours;
date.hoursUTC;
date.hoursUTC = 20;
date.getHours();
date.setHours(5);
date.getUTCHours();
date.setUTCHours(20);
```

Date Arithmetic

To shift time, convert the `Date` object to an epoch time and use normal arithmetic to manipulate it in milliseconds. When you're done, you can write the new value into the existing `Date` with `setTime()`.

TABLE 7-4

Retrieving and Modifying Units of Time

Property	Use	Restrictions
`milliseconds`	Thousandths of a second (0–999)	
`seconds`	Seconds past the minute (0–59)	
`minutes`	Minutes past the hour (0–59)	
`hours`	Hour of the day (0–23)	
`date`	Day of the month (1–31)	
`day`	Day of the week (0–6)	Read-only (no `setDay()` or `setUTCDay()`, implicit accessors are read-only)
`month`	Month (0–11)	
`fullYear`	Unabridged year (1999 instead of 99)	
`time`	Milliseconds since January 1, 1970 UTC	No UTC version of accessors (this property is always relative to UTC)

This approach is recommended over directly setting units of the date as in the following:

```
date.setDate(date.getDate() + 1); // add one day(?)
```

In some cases, if you use arithmetic with time units, you may be responsible for checking all kinds of boundary conditions such as leap years and different numbers of days in a month. This naïve example looks like it might break when it tries to assign 32 to the day of the month. Thankfully, ActionScript is smart and does the right thing.

However, when the amount of time to be added is measured in a unit of time that might depend on special Gregorian rules, you should let the `Date` object do its magic. For example, you should add years by incrementing `fullYear` instead of by adding 365 days to your `Date`. This value for days a year is not entirely as accurate as using the `fullYear` setter would be, because it doesn't account for leap years.

Execution Time

The function `flash.utils.getTimer()` returns the number of milliseconds since Flash Player was initialized. When it's not necessary to use a `Date` object, you can use `getTimer()`, typically to measure elapsed time while a program is executing.

This can be applied to either measure performance of your own application or the network, or to measure time as interactions take place with the user. Call getTimer() once before the event you wish to measure, and once after, and then take the difference to find the duration of the event.

Formatting a Date

We are accustomed to seeing dates written in lots of different ways. When generating text that includes dates, you want control over how the dates will appear so that you can present date information in the most appropriate way.

You can format dates out-of-the-box with a single method call in a few ways. Date objects include a toString() method like every object, as well as several additional formatting methods, as shown in Table 7-5.

TABLE 7-5

Date Formatting Methods

Method	Includes	Example
toDateString()	Month, day, and year	Sun Jan 28 2007
toDateLocaleString()	Same as toDateString()	Sun Jan 28 2007
toString()	All information, 24h	Sun Jan 28 23:00:00 GMT-0800 2007
toLocaleString()	All information except time zone, 12h	Sun Jan 28 2007 11:00:00 PM
toTimeString()	Hours, minutes, seconds, time zone, 24h	23:00:00 GMT-0800
toLocaleTimeString()	Hours, minutes, seconds, 12h	11:00:00 PM
toUTCString()	All information relative to UTC, 24h	Mon Jan 29 07:00:00 2007 UTC

Although these methods are useful to quickly print date information, you can completely control the format of your dates with just a little bit more work by gluing together the units you can access from the Date object itself, and whatever incidental formatting you need.

The following example uses custom formatting to display any date as MM/DD/YYYY:

```
function dateToMMDDYYYY(aDate:Date):String {
    var SEPARATOR:String = "/";

    var mm:String = (aDate.month + 1).toString();
    if (mm.length < 2) mm = "0" + mm;

    var dd:String = aDate.date.toString();
    if (dd.length < 2) dd = "0" + dd;
```

```
        var yyyy:String = aDate.fullYear.toString();
        return mm + SEPARATOR + dd + SEPARATOR + yyyy;
    }
```

The example pads the month and date with leading zeros where necessary, and it shifts up the month field, which is otherwise zero-indexed, by one.

Summary

- There are three number types in ActionScript 3.0, each with different strengths.
- The value `NaN` represents invalid and unassigned numbers; a `Number` can be neither `undefined` nor `void`.
- Edge cases are handled by special constants in the number classes.
- You can operate on numbers with supported operators.
- Numbers can be typed directly in code in multiple notations.
- Numbers can be converted between numeric types, parsed from strings, and printed into strings.
- The `Math` utility class contains static methods for arithmetic and trigonometry.
- `Math.random()` generates pseudorandom numbers between 0 and 1.
- The `Date` class stores an instant in time.
- Epoch time stores an instant in time as a number.
- You can access and alter a `Date` instance using many units of time.

Arrays

I n this chapter, you'll look at *arrays*. Arrays contain indexed data, like a numbered list of items. By using an array, you can store multiple pieces of data in a single variable, which allows you to group values that should go together. An array also provides methods and properties that let you edit it, search through it, sort it, and operate on its contents. Arrays are key tools in any programmer's toolbox.

Arrays are the first complex data type that you study in Part II, "Core Action-Script 3.0 Data Types." Strings, numbers, Booleans, and the like are all primitive data types — the core building blocks of information that usually contain a single piece of data of a specific type. Complex data types, on the other hand, are composites of the various primitive types.

FEATURED CLASS
`Array`

Array Basics

Arrays are a lot like a numbered list of items. Each item, or *element*, in an array has a location, or *index*. Unlike most numbered lists, indices in an array start at 0 instead of 1. These indices are used to look up elements in the array, as shown in Figure 8-1.

Using the Array Constructor

Let's create the first array and fill it with information. For this, you're going to use the `Array` constructor.

```
var myArray:Array = new Array();
```

Tip

Remember that a constructor is a function that creates an instance of a class. A constructor is always named after the class it constructs. To use a constructor, use the `new` keyword followed by the constructor name — for example, `new Array()`. ∎
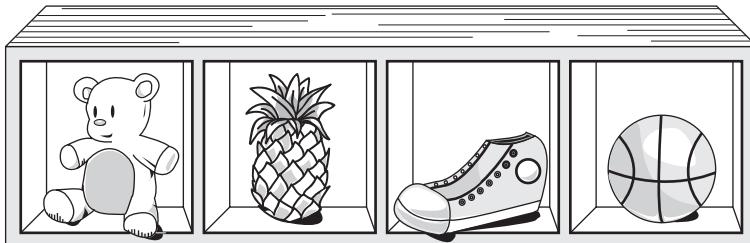
FIGURE 8-1

To visualize an array, think of a set of numbered cubbyholes, each containing a single item.

The previous code creates an empty array — an array that contains no items. Calling the `Array()` constructor with no arguments is the simplest way to make a new array. But ActionScript provides two additional ways to use the constructor. If you want to create an array and assign values to it at the same time, just pass those values to the constructor:

```
var myThings:Array = new Array("coffee filters", "stapler", "Spin Doctors CD");
```

Here, you create an array of your things and simultaneously fill it with some random stuff you might find in one of your desk drawers. I'm filling this array with strings, but arrays in ActionScript can contain any type of object or even disparate types. Sometimes these are called *mixed* arrays.

```
var time:Date = new Date();
var currentTemp:Number = 56;
var currentConditions:String = "Light Showers";
var weatherReport:Array = new Array(time, currentTemp, currentConditions);
trace(weatherReport);
//Sun Dec 3 17:02:16 GMT-0500 2006,56,Light Showers
```

In the previous example, you used the `trace()` statement to print the contents of the array. This is by far the easiest method for finding out what values are stored in an array. This is covered in more detail in the later section, "Converting Arrays to Strings."

The third form of the `Array` constructor allows you to specify the new array's initial length. To do so, call the constructor passing a single integer value signifying the length of the array. A new array is created with as many unassigned indices as you ask for:

```
var topTen:Array = new Array(10);
// creates an array with 10 spaces.
```

In practice, this ability is not terribly useful. An array's length is dynamic and changes to fit your needs for it.

Caution

If you pass a number to the `Array()` constructor with the intent to add that number to the array, you may be disappointed. A single numeric parameter always creates an *empty* array with a capacity specified by the argument. Because this is mighty confusing, I only use the empty, no-parameter `Array()` constructor, or array literals. ∎

Creating an Array by Using an Array Literal

In ActionScript you can write arrays directly into your code using *array literals*. As discussed in Chapter 2, a literal is a value written directly — immediately — into the code, without a constructor or any kind of initialization, like the number 12.

To write an array literal, list the contents of the array separated by commas and surround the whole thing in a set of square brackets (`[]`). For example:

```
[1,2,3]
```

The previous code is equivalent to this:

```
new Array(1, 2, 3);
```

As you can see, the literal form is more compact. Also, you might notice that commas are the way we naturally list things in English. A shopping list might have bananas, coffee, chocolate, and milk. Code imitates life again.

Simply creating an array isn't very exciting unless you plan to do something with it. Let's assign an array literal to a variable:

```
var fibonacci:Array = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55];
```

Likewise, you can create a new empty array by using an empty array literal:

```
var a:Array = [];
```

Literals are a quick and convenient way to create arrays when you know their contents.

Note

Thankfully, ActionScript isn't picky about how you space or terminate your array literals. If you leave a trailing comma in a literal like `[1, 2, 3,]` it will be ignored. Likewise, feel free to put as much or as little whitespace between delimiters as you'd like. ∎

Referencing Values in an Array

Now that you've created some arrays, let's look at how you can access the values they contain. An array's values are stored in slots, or indices. Each index is numbered starting from 0 and counting up to one less than the length of the array. (Try it: if you have four pencils, you would count them: 0, 1, 2, 3. Because the first index of an array is always 0, the last index is always *length* − 1). To look up a value in an array, you'll use the array variable followed by the *array access operator* (`[]`) containing the index you want. The array access operator tells the array to return a specific element at the index you provide:

```
var animals:Array = ["newt", "vole", "cobra", "tortoise"];
trace(animals[2]); //cobra
```

By writing `animals[2]`, you're asking for the item stored at index 2 in the array. An indexed position in an array is no different from a variable with a name: you can assign values to it when it's written on the left side of an assignment:

```
animals[0] = "salamander";
trace(animals); //salamander,vole,cobra,tortoise
```

Finding the Number of Items in an Array

To find out how many elements are contained within an array, you can check the `length` property of the array, as shown in Example 8-1. `length` is a read-only property. You can only set the length of an array by changing its contents, or specifying the length in the `Array()` constructor.

EXAMPLE 8-1 `http://actionscriptbible.com/ch8/ex1`

Array Length

```
var veggies:Array = new Array();
trace(veggies.length); //0
veggies = ["corn", "bok choy", "kale", "beet"];
trace(veggies.length); //4
```

Remember that indices count up from 0, so the length is always one more than the last index in the array. In other words, `veggies[3]` is the last element in the list, whereas `veggies[4]` is undefined.

Converting Arrays to Strings

When working with arrays, it's often helpful to look at a snapshot of the contents. By this point, you're probably familiar with the `toString()` method. This is implicitly called when you use an array where a string is expected. You have already been using this with `trace()`, which prints out its arguments. `Array`'s `toString()` method renders it as a comma-separated list of values.

```
var names:Array = ["Jenene", "Josh", "Jac"];
trace(names); //Jenene,Josh,Jac
```

The `Array` class provides another way to display its values as a string. By using the `join()` method, you can provide your own delimiter to separate the array's contents:

```
var menu:Array = ["eggplant parmesan", "chocolate", "fish tacos"];
trace("For dinner we're having " + menu.join(" and ") + ". Yum!");
```

```
//For dinner we're having eggplant parmesan and chocolate and fish
//tacos. Yum!
```

Here the commas are replaced by the word "and," making the sentence read a little more naturally.

Tip

To create a string with no spaces or other characters between the array values, use the empty string:
`join("")`. ■

Adding and Removing Items from an Array

Adding elements to an array one index at a time is nice, but as the length increases, it can be tricky to keep track of all those indices. Luckily, you won't have to! The `Array` class provides methods for adding items to and removing items from your array without regard for their index numbers. This enables you to collect objects in a list without a care and to use your array as a *stack* or a *queue*.

Appending Values to the End of Your Array with concat()

You've learned how to create individual arrays, so let's look at how you might combine the contents of one array with another array. You might try to do this by adding the two arrays together:

```
var a:Array = [1,2,3];
var b:Array = [4,5,6];
var c:Array = a + b;
```

Running this code results in the following runtime error:

```
TypeError: Error #1034: Type Coercion failed: cannot convert "1,2,34,5,6"
to Array.
```

What's happening here? Well, the two arrays are complex data types, so they're not appropriate values to use with the + operator. Instead, the compiler thinks you want to add two `Strings`, because the + operator works on two strings. It uses `toString()` to coerce the `Arrays` into `Strings` `"1,2,3"` and `"4,5,6"`. Then it concatenates the strings, resulting in `"1,2,34,5,6"`, and tries to assign that back to `c`, which is expecting an `Array`. Only here do you actually get an error, because Flash Player won't convert that `String` into an `Array`. Regardless, this is not what you wanted to do at all.

Tip

A Type Coercion Error usually means that there has been an attempt to assign a value to a variable expecting a different type — for example, trying to assign a `Date` object to a variable of type `Array`. ■

Instead of adding the two arrays together using the plus sign, use the `concat()` method, short for *concatenate*. This method creates a new array starting with its own contents and followed by any number of objects you specify as parameters. If you pass another array as a parameter, the contents of the

array, rather than the array object itself, are added. Example 8-2 adds a to b and throws on a couple of extra values for fun.

Concatenating Arrays

```
var a:Array = [1,2,3];
var b:Array = [4,5,6];
var c:Array = a.concat(b, 7, "eight");

trace(a); //1,2,3
trace(b); //4,5,6
trace(c); //1,2,3,4,5,6,7,eight
```

Notice that the original arrays, a and b, remain unchanged during this operation. The concat() method is nondestructive: it returns a new array that is based on the original but doesn't affect the original array.

Applying Stack Operations push() and pop()

ActionScript's Array has methods that let it act as one of several data structures, or sets of data organized in an efficient way. One such data structure is a *stack*. A stack is a list of data that is maintained in a first in, last out (FILO) manner. In other words, the first thing that goes in is the last thing that comes out.

It might be easiest to think of a stack as a Pez dispenser. Candy loaded in the top gets pushed down onto the stack. When you lift the top of the dispenser, the piece of candy that was most recently loaded pops off the top. The Array class provides methods that do the same thing with its elements.

As with the Pez dispenser, values can be pushed onto the end of an array using push() and popped off the end using pop(). If you think of your arrays as horizontal (left to right) and your stacks as vertical, imagine tipping over the Pez dispenser/stack to the right. You push the pieces of candy "down" to the left, and you pop them off the top at the right. The end of the array is the top of the stack. The push() and pop() methods, rather than creating a new array, modify their own contents. Let's look at this in action:

```
var pez:Array = new Array();
pez.push("cherry");
pez.push("orange");
pez.push("lemon");
pez.push("grape");
trace(pez); //cherry,orange,lemon,grape

var candy:String = pez.pop();
trace(candy); //grape
trace(pez); //cherry,orange,lemon
```

Stacks are useful when your program needs to deal with new information as it comes in but retain the older information to refer to it later. Imagine an alert system that displays a current threat level as a color. When the alert level rises, it might go from green to yellow and return to green when the threat has passed.

The push() method takes one or more objects as parameters and adds them all in order to the end of the array. That is, the objects are added to the last index of the array, making these two statements identical:

```
myArray.push(myValue);
myArray[myArray.length] = myValue;
```

Whether you choose to use a formal stack structure or not, the push() method is a quick and useful way to add elements to the end of your array without being bothered to keep track of indices.

The pop() method, conversely, takes no arguments. It removes and returns the last element of the array.

Applying Queue Operations shift() and unshift()

Similar to stacks are another type of data structure called *queues*. Queues are lists that work in a first in, first out (FIFO) fashion. This could be related to a line of people waiting for tickets at the movie theater. The first person in line is the first person to get a ticket. The next person in line is the next to buy, and so on. Each time a person buys a ticket, he leaves the line and the rest of the line shifts forward one to fill his space.

Arrays can also "push" elements to the end and then "shift" them off the front using, you guessed it, the shift() method. Example 8-3 creates the movie queue example using this technique.

EXAMPLE 8-3 http://actionscriptbible.com/ch8/ex3

Shifting and Unshifting Queues

```
var queue:Array = new Array();
queue.push("Anja");
queue.push("James");
queue.push("Will");
trace(queue);  //Anja,James,Will

var person:String = String(queue.shift());
trace(person); //Anja
trace(queue);  //James,Will
```

As you can see, the first person added to the list was the first person shifted off. The shift() method returns the value removed from the array.

If shifting is taking elements off the front of the array, *unshifting* must be adding elements to the front of the array. Keep in mind this is cutting in line, though! Arrays can also call unshift() to add any number of elements to the front of the array just like push() adds elements to the end of an array.

151

```
queue.unshift("Cheater");
trace(queue); //Cheater,James,Will
```

Queues can be useful when you have data that needs to be processed in a first-come, first-served man-
ner, such as if you want to load several images from a server, one at a time. You'll want the first image
to load before moving on to the next, shifting each one until all the images are done loading in the
exact order you specified.

Note
Stack and queue operations are not mutually exclusive. They can be used alone or together in any array. ∎

Slicing, Splicing, and Dicing

In the previous section, you looked at adding to the front or back of your array. You will probably
use these techniques extensively, but every now and then you need the power to edit whole ranges of
values at once.

Inserting and Removing Values with splice()

"Splice" usually means to join two things by interweaving them, such as pieces of film or rope. With
an array, you can splice together two sets of elements using the `splice()` method. When you splice
together film, you always have to destroy a single frame to get a clean transition. With an array, you
also have the option of deleting elements when you splice in new ones.

The `splice()` method takes two required parameters: the starting index, which is where the inser-
tion or deletion will begin, and the delete count, which is the number of elements you want to delete
from the array (0 if you want to insert only). You may also add any number of optional parameters,
which are elements that will be added to the array at the start index. If any elements are removed
from the array, they're returned by the `splice()` method.

Let's look at `splice()` in action:

```
var nobleGasses:Array = ["helium", "neon", "argon", "pentagon", "xenon",
"radon"];
var shapes:Array = nobleGasses.splice(3, 1, "krypton");
//delete 1 element at index 3, and replace it with "krypton".
trace(nobleGasses); //helium,neon,argon,krypton,xenon,radon
trace(shapes); //pentagon
```

Working with a Subset of your Array with slice()

To extract a subset of your array to work on independently of the original array, you can use the
`slice()` method. You may recall the `slice()` method for the `String` class. Slicing works pretty
much the same way for the `Array` class. Just specify the start and end index (not inclusive) of the
slice you want to create. Example 8-4 shows how this works.

EXAMPLE 8-4 `http://actionscriptbible.com/ch8/ex4`

Collected Snippets: Slicing an Array

```
var metals:Array = ["iron", "copper", "gold", "silver", "platinum", "tin",
"chrome"];
var preciousMetals:Array = metals.slice(2,5);
trace(preciousMetals); //gold,silver,platinum
```

The `slice()`method also accepts negative numbers, which count from the end of the array rather than the beginning.

```
var metals:Array = ["iron", "copper", "gold", "silver", "platinum", "tin",
"chrome"];
var canMakingMetal:Array = metals.slice(-2,-1);
trace(canMakingMetal); //tin
```

Iterating through the Items in an Array

When working with arrays, you'll often want to perform an action using every element in the array. So far, you've looked at ways to create, add, and remove elements in an array; now let's look at how you can work with all of them.

Using a for Loop

The most old-school way to access all the items in an array is the venerable `for` loop. I covered repeating loops in Chapter 2; now you get a chance to put them into action.

When working with arrays, you're likely to see a `for` construct with this form:

```
for (var i:int = 0; i < menuItems.length; i++) {
  makeMenuItem(menuItems[i]);
}
```

Let's look at what's going on here:

- `for` — The `for` keyword specifies a `for` loop.
- `var i:int = 0` — Create a new integer and set the initial value to zero. Frequently named `i` for "iteration" or "index," the loop variable represents the current index in the array.
- `i < menuItems.length` — This specifies that the loop should be repeated while the index is less than the length of the array.
- `i++` — After each loop, the loop variable used the increment operator to count up by one.
- `makeMenuItem(menuItems[i])` — By using the value for `i` as it changes with the array accessor, you can do something with the value of every element in the array.

This is not the only way to loop through an array. Any looping structure can walk over an array. However, now that you've done it the hard way, let's check out how easy visiting items in an array is with `for each..in`.

Using for each..in

This loop was made for arrays! When using a `for each..in` loop, you don't have to keep track of any loop variables, or even access the values of the array by index.

```
for each (var menuItem:String in menuItems) {
  makeMenuItem(menuItem);
}
```

This loop is cleaner and less prone to errors. If you don't care about indices, for example if you're using your array as a list, this method of looping is desirable.

Note
You can put the loop variable's definition inside the parentheses of the loop if you want to. ■

Using the forEach() Method

You just saw how to use a `for each` loop over an array, stepping through the values of an array and executing a block of code for each one. Now see if you can turn this situation inside out in your head. Before, you were acting directly on the array. What if, instead, you ask the array to operate on itself? You could pass the array a function rather than executing code inside the loop. This slight change of perspectives is embodied by the idea of *functional programming*. You'll see more of `Array`'s support for functional programming in the `filter()` and `map()` methods later.

You use the `forEach()` method to make the array apply a function to each of its items:

```
menuItems.forEach(makeMenuItem);
```

The function used for a `forEach()` call takes three parameters. The first is the value of the element in the array, the second is the index of the element, and the third is the array itself. It should match the following function signature:

```
function functionName(element:*, index:int, array:Array):void
```

Let's contrast all these methods of looping through an array in Example 8-5. As always, feel free to modify the code and play with it.

EXAMPLE 8-5 `http://actionscriptbible.com/ch8/ex5`

The forEach() Method

```
package {
  import com.actionscriptbible.Example;
  public class ch8ex5 extends Example {
    public function ch8ex5() {
```

```
    //here's an extra little hack you might find useful
    var menuItems:Array = "About Blog Discography Videos".split(" ");

    trace("---- for loop");
    for (var i:int = 0; i < menuItems.length; i++) {
      makeMenuItem(menuItems[i]);
    }

    trace("---- for each loop");
    for each (var menuItem:String in menuItems) {
      makeMenuItem(menuItem);
    }

    trace("---- Array.forEach()");
    menuItems.forEach(makeMenuItem);
  }

  protected function makeMenuItem(element:String, ...rest):void {
    //just pretend
    trace("making menu item " + element);
  }
}
}
```

Note here that because you don't use the parameters `index` and `array` for anything, I've made them optional by replacing the arguments with `...rest`. That way, you can call `makeMenuItem()` with only one parameter or with the three that `forEach()` uses. Review the `...rest` parameter in Chapter 3, "Functions and Methods."

Searching for Elements

ActionScript 3.0 allows you to search an array for any value using the `indexOf()` and `lastIndexOf()` methods. You may already be familiar with these functions if you've worked with the `String` class, because the methods for the `Array` class work the same way. The methods return the index in the array that contains the specified element. The following code shows how to get the indexes of the elements stored in an array. Notice that "Ocelot" doesn't exist and therefore returns a −1 for its index, indicating that the element could not be found.

```
var cats:Array = ["Cheetah",  "Puma", "Jaguar", "Panther", "Tiger", "Leopard"];
trace(cats.indexOf("Leopard")); //5
trace(cats.indexOf("Ocelot")); //-1
```

Reordering Your Array

Arrays represent ordered sets of data. Frequently, the order is based on the order that you add your data, and in this case you might not even care about the ordering of the contents. Other times, you may want to order data based on factors such as alphabetical name, number of calories in a `Food`

object, fill color of a drawing, or number of points your player has in a video game. Fortunately, ActionScript provides an open-ended solution for sorting the values in an array into whatever order you desire.

Using Sorting Functions

The `Array` class has two methods for sorting contents: `sort()` and `sortOn()`. Using these methods, you can sort structured and unstructured data in simple ways or by providing your own logic. You can achieve the same results with both methods, but `sortOn()` is built for sorting structured data by a specific field.

For both of these methods, you need some data to sort. In Example 8-6, you're going to start with a custom `Book` class that contains the title, author, and publication date of a book. I've created a book-shelf with some books I've enjoyed reading recently. Feel free to substitute your own.

EXAMPLE 8-6 `http://actionscriptbible.com/ch8/ex6`

Sorting Arrays

```
package {
  import com.actionscriptbible.Example;
  public class ch8ex6 extends Example {
    protected var bookshelf:Array;

    public function ch8ex6() {
      bookshelf = new Array();
      bookshelf.push(new Book("Jonathan Strange & Mr Norrell",
        "Susanna Clarke", 2006));
      bookshelf.push(new Book("Anathem", "Neal Stephenson", 2008));
      bookshelf.push(new Book("VALIS", "Philip K. Dick", 1991));
      bookshelf.push(new Book("The Crystal World", "J.G. Ballard", 1966));

      traceBookshelf();
    }

    protected function traceBookshelf():void {
      trace(bookshelf.join("\n"));
    }
  }

}
class Book {
  public var title:String;
  public var author:String;
  public var year:int;

  public function Book (title:String, author:String, year:int) {
    this.title = title;
    this.author = author;
```

```
    this.year = year;
  }

  public function toString():String {
    return '"' + title + '", ' + author + ' (' + year + ')';
  }
}
```

You'll add onto this example in the method `ch8ex6()`, after the data has been created. So far you should simply see the bookshelf in the order you created it:

```
"Jonathan Strange & Mr Norrell", Susanna Clarke (2006)
"Anathem", Neal Stephenson (2008)
"VALIS", Philip K. Dick (1991)
"The Crystal World", J.G. Ballard (1966)
```

Now armed with a set of data, you can explore the sorting functions, starting with the `sort()` method. Using `sort()` with no additional parameters sorts the items in the default manner: alphabetically based on the object's string value. Because you've defined the `toString()` method for the `Book` class to print the book title, then author, then date, you should see the list sorted alphabetically by title. This isn't because the code is sorting by the title per se, but merely because those letters come first in the string representation of the book. The following code:

```
bookshelf.sort();
```

reorders the array to produce this order:

```
"Anathem", Neal Stephenson (2008)
"Jonathan Strange & Mr Norrell", Susanna Clarke (2006)
"The Crystal World", J.G. Ballard (1966)
"VALIS", Philip K. Dick (1991)
```

Notice that the `sort()` method is destructive: it makes changes to the array directly.

Now let's try creating a custom sorting function for this data. To specify your own ordering, you simply have to give the `sort()` method a function. This function compares any two items and tells the sort algorithm which should go first of the two. The comparison functions that you write for the `sort()` method should follow this signature:

```
function sortFunction(valueA:*, valueB:*):Number
```

where *valueA* and *valueB* are two arbitrary values from the array. You'll want to set up your function to return a numeric result based on a comparison between the two values. Use the following rules to determine the results your function should generate:

- If `valueA` should come before `valueB`, return −1.
- If `valueB` should come before `valueA`, return 1.
- If `valueA` and `valueB` are equal, return 0.

You can use this system to sort the items based on any criteria you like. Let's try sorting the books by their date:

```
var byDate:Function = function(a:Book, b:Book):Number {
  if (a.year == b.year) return 0;
  else if (a.year < b.year) return -1;
  else return 1;
}
bookshelf.sort(byDate);
```

Note
You've stored an anonymous function and passed it as an argument to `sort()`. You can also easily pass a method in the current scope or even use an inline function directly within the argument list. ∎

The preceding function compares the `year` property of each `Book` object to see which one comes first, thus sorting the list from oldest to newest:

```
"The Crystal World", J.G. Ballard (1966)
"VALIS", Philip K. Dick (1991)
"Jonathan Strange & Mr Norrell", Susanna Clarke (2006)
"Anathem", Neal Stephenson (2008)
```

You can affect the sort behavior by adding optional sort flags. These are stored as static constants in the `Array` class. To add these options, pass them in separated by *bitwise OR* (|) operators after your sort function. To learn why you do this, check out Chapter 13, "Binary Data and Byte Arrays." The following code sorts the elements in the array numerically (rather than using the string equivalent of the date, which is the default) and descending from the highest to lowest year.

```
bookshelf.sort(byDate, Array.NUMERIC | Array.DESCENDING);
```

produces the following result:

```
"Anathem", Neal Stephenson (2008)
"Jonathan Strange & Mr Norrell", Susanna Clarke (2006)
"VALIS", Philip K. Dick (1991)
"The Crystal World", J.G. Ballard (1966)
```

You can pass five such optional flags to either `sort()` or `sortOn()`:

- `CASEINSENSITIVE` — Normally, sorting is case-sensitive. Using this flag makes the search ignore the case of letters. You probably want this for most string sorting.

- `DESCENDING` — Using this causes the array to be sorted from highest to lowest.

- `NUMERIC` — If you're sorting only numbers, use this flag. Otherwise, numbers are converted to their string equivalent before sorting occurs. This can be really tricky! For instance, as strings, "2009" is greater than "1000000" because the "2" character is higher than the "1" character. Make sure to enable numeric comparisons to avoid these errors! In this case, writing your own comparison function that explicitly compares years prevents this pitfall.

- `RETURNINDEXEDARRAY` — This flag causes the sort function to return a sorted array without affecting the contents of the original array on which the sort method was called. Moreover, this array contains the indices of which elements should go where, rather than the elements themselves.

- UNIQUESORT — When this flag is set, the sorting method aborts and returns 0 if any two elements are found to be equal.

You looked at using sort() to compare elements using a function; now let's look at the sortOn() method, which allows you to automatically compare similar properties of two composite data types. Rather than taking a sorting function as an argument, the sortOn() method takes an array with one or more properties to use as sorting criteria. If a single property is passed in, the elements are sorted by that criterion. If more than one property is passed in, the elements are sorted primarily by the first property and then secondarily by the second property, and so on. Omitting the first parameter sorts the list alphabetically, just like the sort() method. Also, the optional parameters can be applied in the same manner as they are with the sort() method.

```
bookshelf.sortOn(["year"], Array.NUMERIC);
bookshelf.sortOn(["author"], Array.CASEINSENSITIVE);
```

The first line sorts by year and is identical to the preceding byDate() example. The next, as you guessed, sorts by author. Try it yourself.

Flipping the Order of Your Array Using Reverse()

One of the quickest ways to reorder your array is to reverse it — that is, flip the order so that the last is first and vice versa. To do this, use the reverse() method on your array:

```
var myArray:Array = [12,6,17,4,39];
myArray.reverse();
trace(myArray); //39,4,17,6,12
```

This reorders the contents within the array on which it is called without creating a new array.

Applying Actions to All Elements of an Array

You were briefly introduced to functional programming in "Iterating through the Items in an Array" earlier. In this section you see other ways that you can treat an array as a collection of values that you apply functions to. You can replicate all these operations with looping code, but directly applying functions, tests, transformations, and filters to whole lists of data is a powerful, dare I say beautiful, technique.

All these methods take functions as parameters with the same signature as the forEach() method shown earlier. There is one subtle difference: the every(), some(), and filter() methods require functions that return a Boolean rather than void.

```
function functionName(element:*, index:int, array:Array):Boolean
```

Conditional Processing with every(), some(), and filter()

The first methods you'll learn about are designed to allow you to see what values of an array fulfill a certain condition. The every() method iterates over every element, applying the function passed into it until one of the elements causes the function to fail, returning false. The some() method

works the same way but, in its case, it stops when a value of `true` is reached. If the "stopper" value is reached, either method returns the last value processed. If the end of the array is reached without interruption, `every()` returns `true` (every value met the test) and `some()` returns false (none of the values met the test).

Let's check out an example. Consider the array and filtering functions shown in Example 8-7.

EXAMPLE 8-7   `http://actionscriptbible.com/ch8/ex7`

Array Filtering Functions

```
package {
  import com.actionscriptbible.Example;
  public class ch8ex7 extends Example {
    public function ch8ex7() {
      var myArray:Array = [1,2,3,4,5];

      trace("---- is every value less than three?");
      trace(myArray.every(lessThanThree));

      trace("---- are some of the values less than three?");
      trace(myArray.some(lessThanThree));

      trace("---- are all values less than ten?");
      trace(myArray.every(lessThanTen));

      trace("---- are some values more than ten?");
      trace(myArray.some(moreThanTen));

      trace("---- which numbers are less than three?");
      trace(myArray.filter(lessThanThree));
    }
    private function lessThanThree(elem:*, i:int, a:Array):Boolean {
      trace(elem);
      return (elem < 3);
    }
    private function lessThanTen(elem:*, i:int, a:Array):Boolean {
      trace(elem);
      return (elem < 10);
    }
    private function moreThanTen(elem:*, i:int, a:Array):Boolean {
      trace(elem);
      return (elem > 10);
    }
  }
}
```

You have an array with a few integers in it. Each of these functions simply traces the element that it's evaluating and then returns whether the element is "less than three," "less than ten," or "more than

ten," respectively. Now let's see what happens when you apply some of these methods, starting with
every(lessThanThree):

```
trace(myArray.every(lessThanThree));
1
2
3
false
```

Checking every() evaluates each item through three and then fails and returns false because
3 < 3 is false. Now let's try the some() method.

```
trace(myArray.some(lessThanThree));
1
true
```

The some() call doesn't even get as far as three. It stops after the first true value (1 < 3). If it's
true for at least one value, that makes it true for some of the values, right? Now let's try some
examples where the functions run through every item:

```
trace(myArray.every(lessThanTen));
1
2
3
4
5
true
```

Every item in the list is less than 10, so they're all evaluated as true. You have to try each one to
make sure there's not a really big number at the end. Let's try again with the some() method with
moreThanTen.

```
trace(myArray.some(moreThanTen));
1
2
3
4
5
false
```

Conversely, all the values are not more than 10, so the some() function executes all the way to
the end.

The filter() method acts similarly to the previous two methods but adds a twist. Instead of search-
ing for a Boolean stopper value, this method processes the entire array and then returns a new array
containing all the elements that resulted in a true:

```
var smallNumbers:Array = myArray.filter(lessThanThree);
trace(smallNumbers); //1,2
```

Getting Results with the map() Method

The map() method takes a slightly different approach than every(), some(), and filter(). You use this method to transform, or map, one list to another. Whereas forEach() just executes a method on every element without returning anything, map() creates a new list from the results of its function applied to every element. Mapping takes in an input list, applies a function, and outputs a new list. The following code demonstrates a map that returns the square of every element.

```
var myArray:Array = [1,2,3,4,5];

function square(elem:*, i:int, a:Array):Number {
  if (isNaN(elem)) {
    return -1;
  } else {
    return elem * elem;
  }
}

var squaredArray:Array = myArray.map(square);
trace(myArray);      //1,2,3,4,5
trace(squaredArray); //1,4,9,16,25
```

You use an if statement to check whether the element is not a number using isNaN() because, of course, you cannot multiply a nonnumber. If the element is a nonnumber, the function returns a −1; otherwise, it returns the number raised to the power of 2. The original array, myArray, is untouched and the new array, squaredArray, contains the results of the getSquare() function when it's applied to each element of myArray. This can be useful when you need to gather results based on a large set of data.

Alternatives to Arrays

As you've seen, Arrays are incredibly versatile. But there are even more ways to use an Array that you haven't explored yet, and you can do similar things with a few other classes.

Associative Arrays

All types of dynamic objects in ActionScript, of which Array is one, allow you to create new variables stored within the object on-the-fly. You can then access these variables using the array accessor and passing a string rather than an index as a reference. These types of objects are known as *associative arrays*. The following code demonstrates an associative array.

```
var lotto:Array = new Array(12, 30, 42, 6, 29, 75);
lotto.description = "This week's lotto picks.";
trace(lotto[3]); //6
trace(lotto["description"]); //This week's lotto picks.
```

Although these types of arrays are possible to create and may have been used frequently in previous versions of ActionScript, I suggest that you use an instance of the Object class, a dictionary, or ideally, a custom class, rather than an array for such applications. Here are some reasons why.

- Objects are designed for this type of use and offer the object initializer syntax.
- They have less overhead than arrays because they do not contain methods for array manipulation.
- Values stored by name may not appear when iterating through all values in an array using a `for` loop.
- Values stored by name are not kept in any particular order, making the concept of an array as an ordered set of data untrue.
- Storing data dynamically in this way is not good object-oriented practice and makes debugging and type checking more difficult.

That's not to say there's anything wrong with associative arrays. But I recommend that you use `Object` or `Dictionary` rather than `Array` for associative arrays. Learn more about these classes in Chapter 10, "Objects and Dictionaries."

Multidimensional Arrays

All the arrays you've looked at so far are single-dimensional arrays. That is, they have one index and represent a linear list of data — hence, one-dimensional. With a multidimensional array, however, you can store values using two or more nested indices, which allow you to create data sets within data sets. Creating a multidimensional array doesn't require new classes or techniques; simply create an array as one of the elements of another array:

```
var grid:Array = new Array(new Array(1,2), new Array(3,4));
//or var grid:Array = [[1, 2], [3, 4]];
trace(grid[0][0]); //1
trace(grid[0][1]); //2
trace(grid[1][0]); //3
trace(grid[1][1]); //4
```

An expression like `grid[0][1]` has two implicit steps. `grid[0]` retrieves the first index in the outer array. You find that this, too, is an array and `[1]` retrieves the second item in the inner array.

Suppose you want to store information in two dimensions — such as the locations of the pieces on a chessboard. Well, a chessboard is 8 squares wide by 8 squares high, so it's 8 × 8. To store all the values for this board, you would need 8 arrays, each with a length of 8. Using a two-dimensional array, you could store this 2D grid-like data in one array. Example 8-8 sets up a new chessboard by creating several arrays within one array.

EXAMPLE 8-8   `http://actionscriptbible.com/ch8/ex8`

A Chessboard as a Two-Dimensional Array

```
package {
  import com.actionscriptbible.Example;

  public class ch8ex8 extends Example {
    private const k:String = "King";
    private const q:String = "Queen";
    private const b:String = "Bishop";
```

163

```
    private const n:String = "Knight";
    private const r:String = "Rook";
    private const p:String = "Pawn";
    private const _:String = "empty";

    public function ch8ex8() {

      var chessBoard:Array = [
        [r,n,b,q,k,b,n,r],
        [p,p,p,p,p,p,p,p],
        [_,_,_,_,_,_,_,_],
        [_,_,_,_,_,_,_,_],
        [_,_,_,_,_,_,_,_],
        [_,_,_,_,_,_,_,_],
        [p,p,p,p,p,p,p,p],
        [r,n,b,q,k,b,n,r]
      ];

      trace("Piece at (0,2) :", chessBoard[0][2]);
      trace("Piece at (7,4) :", chessBoard[7][4]);
      trace("Piece at (4,3) :", chessBoard[4][3]);
      trace("Piece at (7,7) :", chessBoard[7][7]);
    }
  }
}
```

The example will display:

```
        Piece at (0,2) : Bishop
        Piece at (7,4) : King
        Piece at (4,3) : empty
        Piece at (7,7) : Rook
```

What you've done here is to create an 8 × 8 grid of data representing each space on a chessboard. Using constants to represent each piece and the array literal shorthand, you've added all the pieces to the board. It bears repeating that this is simply an array of rows, each of which is an array of columns. But it works out such that you can think of the indices as y and x coordinates. When you say `chessboard[7][4]`, what you're really doing is saying get the value at the eighth index of the `chessboard` array, which in this case is another array, a row. Then you're selecting the fifth index of the array that is returned.

You can use arrays like this one, or create even more complex 3- or 4- or 17-dimensional arrays whenever you need to deal with a complex geometric or cross-referential set of data.

Another name for a multidimensional array is a matrix. Matrices are used so frequently in graphics programming that several matrix classes are included in ActionScript 3.0, but these are special-purpose matrices of fixed sizes. You can't use these for generic multidimensional data.

Amazing Properties of Arrays

Before we say farewell to `Array`, let's recap a few of its fantastic properties and make some quick observations on how ActionScript 3.0 arrays compare to similar data types in other languages. Arrays are possibly the most versatile tool in ActionScript.

- Arrays are dynamically sized. You never have to do anything to allocate more memory for them or free memory they're not using.

- Arrays are always mutable. You can always create, remove, update, and delete values from them.

- Arrays support list operations like `push()` and `pop()`, and functional programming like `filter()` and `map()`.

- Arrays are mixed. You can put multiple types into an array, even if they're not the same type.

- Arrays can be sparse, meaning that you can have a gap in the indices of an array. This can be really helpful or really dangerous. Use with caution.

```
var arr:Array = ["zero", "one", "two"];
arr[5] = "five";
trace(arr); //zero,one,two,,,five
```

- Arrays can be associative, but you should probably leave that up to `Objects`, as mentioned earlier.

In C++ and Java, dynamic arrays that can resize and implement list operations like this are called *vectors*. In ActionScript 3.0, arrays do all this stuff, and a vector means something slightly different. See Chapter 9, "Vectors," to find out more.

Summary

- Arrays are ordered lists of data that use index numbers to look up values called elements.

- You can create new arrays by using the `Array` constructor or array literals `[]`.

- Create strings from array values using the `toString()` and `join()` methods.

- Access values within your array by using the array accessor operator (`[]`).

- Searching for a value within your array is easy with the new `indexOf()` method.

- Perform actions on every element in your array using `forEach()`, `map()`, `filter()`, `every()`, or `some()`.

- Insert values into your array or remove smaller chunks with `slice()` and `splice()`.

Vectors

V ectors are a new addition to ActionScript 3.0 for Flash Player 10 and later. They can help you write fast, efficient, and concise code where you would otherwise use an array.

Vector Basics

A *Vector* is an optimized kind of `Array` that is made to store and retrieve instances of the same class. It behaves almost identically to an `Array`, but the key difference is that it stores only one type of object, as you can see in Figure 9-1.

In the original `Array`, you can store all kinds of objects together. In a `Vector`, you store only one kind. In the preceding example, the `Vector` only stores instances of `Shoe`. In this chapter, you'll see when and why this can be useful. I'll also cover the other minor differences between the two data types.
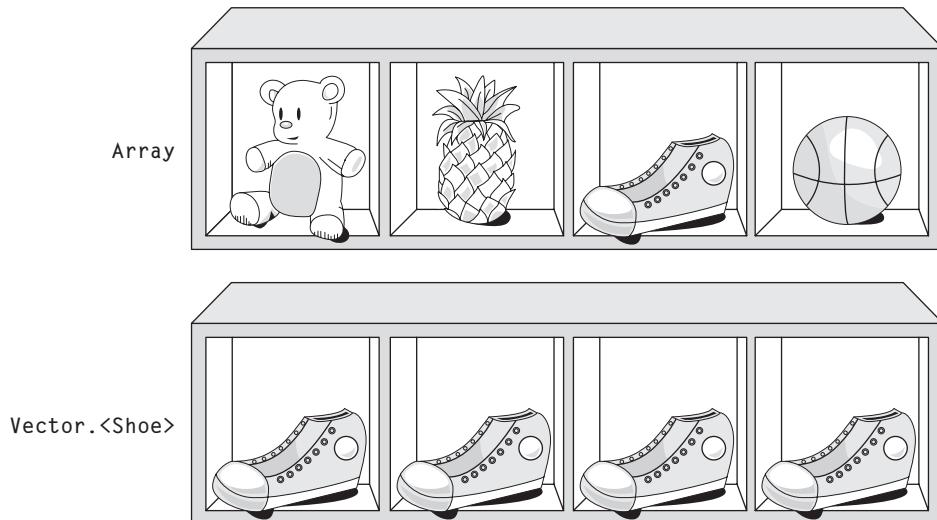
Version
FP10. Vector is available when compiling SWFs for Flash Player 10 and above. ∎

Vectors are only available on Flash Player 10 and up. If you're reading this book sequentially, this is the first time you will have seen a whole class that is only available in certain versions of Flash Player.

FIGURE 9-1

An Array and a Vector compared

Before I examine the when, why, and how of using Vectors, I'll take a moment to answer the next question:

Why Do We Need Another Datatype?

In many ways, ActionScript has been evolving backward, from a "modern" dynamically typed, high-level, object-oriented language toward a statically typed language with lower-level constructs. Classes are now sealed, where in ActionScript 1.0 and 2.0 they were dynamic and modifiable at runtime. ActionScript 3.0 added direct access to binary data with the `ByteArray` class and efficient integer storage with `int` and `uint`. It's as if you give your daughter a set of Legos that are bright and easy to use, and then the next year you give her a vintage Erector Set. The Erector Set is decidedly less friendly and a bit dated, but perhaps in some ways it's a bit lighter and capable of making entirely different kinds of structures. There are a few reasons behind this evolution.

One is that some tools are better for some jobs, and it makes sense to give developers more power so that they can build things not otherwise possible. For example, without `ByteArray` and binary sockets, added in ActionScript 3.0, you couldn't write a POP/SMTP mail client in ActionScript, a VNC client, a JPEG encoder, or a million other things.

Another reason for this odd evolution is the unquenchable desire for more performance. As developers do more with Flash Player, demands on it grow, and as its speed boosts, developers can do more with it. Everyone wins. This explains the funny direction of evolution. Dynamic typing and high-level objects are built on lots of code and require lots of extra support in Flash Player — in other words, overhead, taxing both memory and processor. Of course, primary types don't require overhead, so they're faster. There's always some kind of trade-off between functionality and performance. It's that old "no such thing as a free lunch" rule. So what better way to feed the power-lust of developers than by letting them use simpler objects when they don't need the extra functionality?

For example, when you keep a class sealed, by omitting the `dynamic` keyword on the class definition or by default (see Chapter 4, "Object Oriented Programming,") Flash Player knows you won't be adding random fields to the class at runtime. It is guaranteed a complete picture of the layout of a class's variables, and with this guarantee, it can optimize the layout of the class instances in memory. You lose the ability to muck with class definitions at runtime, but you gain a big performance boost.

Tip

The more you use dynamic types, the less performance you'll see overall. That's why examples always use static types. ■

Array: More Functionality Than You Require

In ActionScript 3.0, the `Array` class provides a lot of really cool possibilities. To review them, check out the end of Chapter 8, "Arrays." But, you guessed it, there are a few of those amazing properties of Arrays that maybe we could do without — properties that make an `Array` a pretty complicated object. Throw away some of these properties, and you can make a simple, fast, efficient data type instead: a `Vector`.

Note

Oddly, in languages that have both an `Array` and a `Vector`, it is usually the `Vector` that has more functionality — usually the ability to change its size at runtime. However, in ActionScript, `Vector` is more limited. ■

The essential property of an `Array` that a `Vector` does away with is its ability to store mixed types. Nine times out of ten, you use `Array`s as lists to accumulate objects of the same type, such as when iterating over `DisplayObject`s on the stage, accumulating `XML` nodes in a document, or creating a buffer of `int`s. In all these cases, you know exactly what type is going to go into the array, and you have no need to store other types. However, as long as it's an `Array`, Flash Player doesn't really know what type any of the elements are until it looks. Any time you assign an untyped object into a typed variable, Flash Player must either match the types or coerce the object into the desired type — and every element of an `Array` is stored untyped. So any time you use an element from an `Array` in a typed context, Flash Player has to figure out these types — a simple operation, but when repeated hundreds or thousands of times while processing large arrays, one that can have a cumulative effect.

Note

If you care to dig deeper, knowing the inner workings of the ActionScript Virtual Machine (AVM) can help you better understand how code behaves, and it can even help you write more efficient code. An excellent introduction to the guts of AVM2 is a presentation by Gary Grossman titled "ActionScript 3.0 and AVM2: Performance Tuning." You can find the slides at `http://bit.ly/grossman-avm2-presentation`. ■

The other `Array` property you benefit from simplifying away is its sparseness — the ability to add items at any index at any time, allowing gaps. In a `Vector`, you can only mess with indices up to the length of the `Vector`, preventing gaps from forming. So what does this simplification gain you? If you know the size of an object in memory (which you do, since you know its type), and you know that you will sequentially store n of them without gaps, this makes finding and organizing the objects painfully simple: just line them up in memory! Need to retrieve the `int` at index i from a `Vector` of `int`s? All Flash Player has to do is look ahead $4i$ bytes, because an `int` is 4 bytes wide and the `Vector` is packed tight with them. Again, this is exactly the same kind of old-school simplicity from

back in the day: C arrays are just blocks of memory chopped up into chunks the size of the type being stored. This simplification makes `Vectors` not only faster on the whole, but more efficient users of memory.

Note

You are still allowed to store null instances in a Vector, so although it's not completely identical, this is a perfectly legal way to have gaps. ∎

Vectors: Arrays with Benefits

There you have it. `Vector` is a super-efficient alternative to `Array` for when you don't need mixed content. Although we got from `Array` to `Vector` by taking away features of the data type, don't think of `Vector` as less powerful. Besides making it faster and smaller, the stricter definition of `Vector` helps the compiler and runtime environment check your code, and it can even help you write less code!

The first trade-off that you saw with static types is that they can be faster. But they can also be safer, for the same reasons that using typed variables is safer than using untyped variables. The type system can check that you don't assign incompatible types, either at compile time or at runtime.

```
var ourguy:Person = new Person("Roger");
ourguy = "Xander"; //error!

var classroom:Vector.<Person> = new Vector.<Person>();
classroom[0] = new Person("Betty");
classroom[1] = "Cesar"; //error!
```

In the previous example, you can't assign a `String` to a variable of type `Person`. In the same way, you can't assign a `String` to a `Vector` of `Persons`. When you use vectors, the type system can ensure the type safety of your `Vector` during assignment, retrieval, and insertion. Say you have both an `Array` and a `Vector` of `Person` instances:

```
var peopleArray:Array = new Array();
var peopleVector:Vector.<Person> = new Vector.<Person>();
var person:Person;
var nonPerson:DisplayObject;

//assignment
peopleVector[0] = person; //OK
peopleVector[0] = nonPerson; //Compiler error (good!)
peopleArray[0] = nonPerson; //No error but not what we want (bad!)

//retrieval
person = peopleVector[0]; //OK
nonPerson = peopleVector[0]; //Compiler error (good!)
nonPerson = peopleArray[0]; //Runtime error (not as good.)

//retrieval: range checking
person = peopleVector[0]; //OK
person = peopleVector[34000]; //Runtime error (not bad)
person = peopleArray[34000]; //No error, just sets it to null (bad)
```

```
          //insertion
          peopleVector.push(person); //OK
          peopleVector[peopleVector.length] = nonPerson; //Compiler error (good)
          peopleVector.push(nonPerson); //Runtime error (not bad)
```

It's quite intuitive how this works: the Vector knows the type of element that it is meant to contain, so it can check against this type when you try to assign and insert elements into it, and it can return elements of the correct type when you retrieve from it, as shown in Example 9-1.

EXAMPLE 9-1 http://actionscriptbible.com/ch9/ex1

Working with Vectors

```
package {
  import com.actionscriptbible.Example;
  public class ch9ex1 extends Example {
    protected var peopleVector:Vector.<Person>;
    protected const roger:Person = new Person("Roger");
    protected const sabrina:Person = new Person("Sabrina");
    protected const tawnie:Person = new Person("Tawnie");
    protected const usman:Person = new Person("Usman");
    protected const victoria:Person = new Person("Victoria");

    public function ch9ex1() {
      peopleVector = new Vector.<Person>();
      peopleVector.push(roger, sabrina, tawnie, usman, victoria);
      for each (var person:Person in peopleVector) {
        roger.addFriendSymmetric(person);
      }
      sabrina.addFriendSymmetric(tawnie);
      var tawniesFriend:Person = new Person("Mary Alice");
      tawnie.addFriendSymmetric(tawniesFriend);
      var nixon:Person = new Person("Richard Nixon");

      trace("is Mary Alice connected to Victoria?");
      trace(tawniesFriend.isConnectedTo(victoria)); //true
      //Mary Alice -> Tawnie -> Roger -> Victoria
      trace("is Roger connected to Richard Nixon?");
      trace(roger.isConnectedTo(nixon)); //false
    }
  }
}

import flash.utils.Dictionary;
class Person {
  public var name:String;
  private var friends:Vector.<Person>;

  public function Person(name:String) {
    this.name = name;
    friends = new Vector.<Person>();
  }
```

EXAMPLE 9-1 *(continued)*

```
  public function addFriend(newFriend:Person):void {
    friends.push(newFriend);
  }
  public function addFriendSymmetric(newFriend:Person):void {
    addFriend(newFriend);
    newFriend.addFriend(this);
  }
  public function isFriendsWith(person:Person):Boolean {
    return (friends.indexOf(person) != -1);
  }
  public function isConnectedTo(toFind:Person, met:Dictionary = null):Boolean {
    var ret:Boolean = false;
    if (met == null) met = new Dictionary(true);
    for each (var contact:Person in friends) {
      if (!(contact in met)) {
        met[contact] = true;
        if (contact == toFind) {
          return true; //found them!
        } else {
          ret ||= contact.isConnectedTo(toFind, met);
        }
      }
    }
    return ret;
  }
}
```

In this example you can use a Vector to store friends, friends of friends, and so on, using a recursive search to determine if any two people are connected. You can use code like this to implement a social network. Note that all the methods so far are exactly like their Array counterparts.

Fixed-Size Vectors

There is just one more feature of Vectors that can make them slightly faster. You can set a vector to be, or create a vector as, a fixed size. Fixed-size Vectors don't allow operations that modify their length. Methods that nondestructively modify Vectors by returning a modified copy are fine, as long as the fixed-size instance itself is not changed in length. When used on fixed-sized Vectors, methods that modify a Vector's length, like push() and pop(), generate a RangeError at runtime.

You can make a Vector fixed-size by passing true to the fixed parameter of its constructor (at the same time, specifying the size the Vector should be) or by setting its fixed property to true at any time.

```
    var v1:Vector.<Object> = new Vector.<Object>(10, true);
    //creates a fixed-size vector of size 10
    var v2:Vector.<String> = new Vector.<String>();
    v2.push("a", "b", "c", "d");
```

```
        v2.fixed = true;
        trace(v2.length);
        v2.push("e"); //RangeError: v2 is fixed-size now.
```

Fixed-length `Vectors` are useful when you know the ultimate number of elements you'll need. Because `Vectors` are stored in memory as a solid, unbroken block of data, as they grow they may occasionally need to do a relatively costly reallocation of memory. In extremely performance-sensitive applications, you may want to preallocate `Vectors` to the correct size to prevent this.

Generics and Parameterized Types

The ability you just saw of `Vectors` to access elements as the type the `Vector` is meant to hold is an example of *generic programming*. Generic programming adds a layer to the type system of ActionScript, and at the same time a little bit of new syntax to keep things interesting. Generic programming allows you to generalize algorithms and data structures so that they are independent of the type of objects they operate on; they also allow you to make carbon copies of these templates when you swap in a type for them to use.

Vector as a Generic

`Vector` is a *generic* because you always use it with another type. `Vector` doesn't care whether it's used to store `uints`, `Persons`, or even other `Vectors`. Neither does an `Array`, but the difference is that when you create a vector with a specific type, it actually changes the return types and parameter types of its methods to match that type.

For example, if you wrote a `Rolodex` structure to manage `Person` contacts, you might one day realize that it would be perfect to also manage `Company` contacts. Rather than having to write a sub-class or copy of the `Rolodex` class that works for `Company` objects, wouldn't it be great if you could abstract out the code that sorts and organizes the contacts so that it would work with *any* type? Then you could use one `Rolodex` just for people, and one just for companies. Each one would know through the whole interface that it's made to deal with its one specific type. You could use them like so:

```
        var personalDeck:Rolodex.<Person> = new Rolodex.<Person>();
        var randomPerson:Person = personalDeck.getRandomEntry();
        personalDeck.placeCall(randomPerson);
        var corporateDeck:Rolodex.<Company> = new Rolodex.<Company>();
        var randomCompany:Company = corporateDeck.getRandomEntry();
        corporateDeck.placeCall(randomCompany);
```

You would do this by *parameterizing* the types in `Rolodex`. Where originally it would have had these methods:

```
        class Rolodex {
          public function getRandomEntry():Person {}
          public function placeCall(a:Person):void {}
        }
```

after replacing `Person` with a *type parameter*, it would have these methods:

```
        class Rolodex.<T> {
          public function getRandomEntry():T {}
          public function placeCall(a:T):void {}
        }
```

173

You've taken the type `Person`, abstracted it out, and replaced it with the *type parameter* called `T`. This type parameter is almost like a variable name, but it actually holds a type instead of a value. Now any type can take the place of `Person`. You can create separate *invocations* of `Rolodex` that use completely different types. When you create a `Rolodex.<Company>`, you effectively replace all instances of `T` in the interface with `Company`. Now `getRandomEntry()` returns a `Company`, and `place-Call()` takes a `Company`. You've replaced the type parameter with an actual type.

This is why the type checks shown in the previous section work. When you create a `Vector.<Person>`, it uses `Person` instead of `T` in the signatures of all its methods. When you look at the documentation for `Vector` (and let me remind you that owning this book is no excuse to ignore the documentation!), you'll see that the methods of `Vector.<T>` are almost identical to those of `Array`, except that where `Array` methods receive and return elements as `*`, or no type, `Vector.<T>` methods receive and return elements as `T`. So when you create a `Vector.<Person>`, you're using an invocation of `Vector` that replaces the type parameter `T` with the actual type `Person` in the entire interface of `Vector`.

One more thing you should know about the type system and generics: when you specify a type for the template to use, you create a new type that's the combination of the types. A `Rolodex` of `Person`s is one type. A `Rolodex` of `Company`s is another type entirely, and it's *not* compatible, even if `Company` and `Person` are related through inheritance. You use the full expression wherever you need to write the type into the code. For example, if you needed to call all companies in a `Rolodex`, you might create a function:

```
function callAllCompaniesIn(r:Rolodex.<Company>):void {}
```

The type this function accepts is `Rolodex.<Company>`. It won't accept `Rolodex`es containing any other thing — those types are not compatible. When you're referring to a particular invocation of a generic type like this, the type iscalled a *parameterized type*.

There are some interesting and tricky side effects of the way some `Vector` methods are written to become generic, so for both reference and caution's sake I'll examine them in the next section.

No Generics for You

Here comes the funny part. In the `Rolodex` thought experiment, I asked, "Wouldn't it be great if [you could just make this `Rolodex` class a generic]?" Remember? Because one answer is "Maybe, but we'll never know, because you can't."

Up through Flash Player 10.1 at least, `Vector` holds a prized position as the one and only generic in all of ActionScript 3.0. You can't take a sweet data type you created and parameterize the type it uses. Sadly, you will see no parameterized `Tree.<T>`s or `Heap.<T>`s. For now at least, it's a special case that exists for performance tuning.

Although you can build your own data types that use `Vector` internally for performance, you won't be able to parameterize them and take advantage of the no-cast no-cost element access that you get with a generic.

The thing is, you don't really need generics. Judicious use of interfaces makes a lot more sense than a type parameter in most cases. For example, when you think of it, the `Rolodex` example is really terrible. How would a `Rolodex` work that stores emotions? Cupcakes don't have phone numbers ... how would you implement `placeCall()` when `T` is `Cupcake`? In reality, you almost always need certain guarantees about the objects you're dealing with, even if it's something as simple as the ability to compare two of them. Nowhere does Flash Player give you a sort function that works on all

objects. In other words, your ability to do interesting things generically is limited by the things that you're guaranteed you can do with any arbitrary object. In Flash Player, any arbitrary object means "subclasses of `Object`," and `Object` is not endowed with many powers.

In this case, you really don't want to be able to give `Rolodex` any type in the world. You want it to operate on objects that have phone numbers to call and a name to sort. Hey, you have a mechanism for that:

```
interface IContact {
  function get phoneNumber():Number;
  function get fullName():String;
  function get lastName():String;
}
```

The `Rolodex` example is far from generic. It should be implemented to only operate on objects that have certain properties. The perfect device for holding objects to a certain contract is an interface. So `Rolodex` should not be generic. Its interface should be written around an `IContact` interface that allows it to do the interesting things a `Rolodex` does. Internally, you're more than welcome to use a `Vector.<IContact>` to make `Rolodex` fast and efficient! (See Example 9-2.)

You should use an interface or a common superclass in lieu of generics to keep your classes abstract. The only drawback you'll see is that you may have to do some potentially dangerous upcasts.

EXAMPLE 9-2 `http://actionscriptbible.com/ch9/ex2`

The Rolodex Class

```
package {
  import com.actionscriptbible.Example;
  public class ch9ex2 extends Example {
    public function ch9ex2() {
      var rolodex:Rolodex = new Rolodex();
      rolodex.addEntry(new Person("Roger", "Braunstein", 7185555555));
      rolodex.addEntry(new Person("Simon", "Bolivar"));
      rolodex.addEntry(new Person("Shimon", "Peres"));
      trace("B entries:", rolodex.getEntriesUnderLetter("B"));
      var firstB:Person = Person(rolodex.getEntriesUnderLetter("B")[0]);
      //note that we still have to upcast because this returns an IContact.
    }
  }
}

class Rolodex {
  protected var entries:Vector.<IContact>;
  public function Rolodex() {
    entries = new Vector.<IContact>();
  }
  public function addEntry(contact:IContact):void {
    entries.push(contact);
    entries = entries.sort(function(a:IContact, b:IContact):Number {
      return (a.lastName.toLowerCase() < b.lastName.toLowerCase())? -1 : 1;
    });
  }
```

EXAMPLE 9-2 *(continued)*

```
  public function getEntriesUnderLetter(ltr:String):Vector.<IContact> {
    ltr = ltr.toLowerCase();
    return entries.filter(function(entry:IContact,...rest):Boolean {
      return (entry.lastName.charAt(0).toLowerCase() == ltr);
    });
  }
  public function getRandomEntry():IContact {
    return entries[Math.floor(Math.random() * entries.length)];
  }
  public function placeCall(contact:IContact):void {
    trace("calling", contact.phoneNumber);
  }
}

interface IContact {
  function get phoneNumber():Number;
  function get fullName():String;
  function get lastName():String;
}

class Person implements IContact {
  private var _phoneNumber:Number;
  private var _lastName:String;
  private var _firstName:String;
  public function Person(first:String, last:String = "", number:Number = 0) {
    _firstName = first;
    _lastName = last;
    _phoneNumber = number;
  }
  public function get phoneNumber():Number {return _phoneNumber;}
  public function get fullName():String {return _lastName + ", " + _firstName;}
  public function get lastName():String {return _lastName;}
  public function toString():String {return "[Person " + fullName + "]";}
}
```

Even though I've designed `Rolodex` to operate on `IContacts`, and it uses a `Vector` for storage internally, you don't have the leisure of returning contacts as type `Person` or `Company`; you still have to upcast.

Generics are best suited for optimized storage data types, which don't depend in any way on the contents or structure of the data. While you can't create your own, `Vector`, just like `Array`, is incredibly useful and flexible.

Generic Methods of Vector

`Vector` methods correspond almost exactly to `Array` methods, substituting the type parameter (T) wherever elements are concerned — where `Array` would use `*`. This makes it easy to switch over code that uses `Arrays`. Without covering every method, let's look at those that are a bit tricky.

```
concat(... args):Vector.<T>
push(... args):uint
```

Because these methods can take a list of elements to add to the Vector all at once, they use a variable argument list instead of type T. As you saw earlier, this means that the arguments are not type-checked at compile time. They are still checked against the type parameter T at runtime.

```
every(callback:Function, thisObject:Object = null):Boolean
some(callback:Function, thisObject:Object = null):Boolean
filter(callback:Function, thisObject:Object = null):Vector.<T>
```

All these methods act the same as their Array counterparts, except that now you can expect to iterate over objects of type T, so the signature of the callback function must be:

```
function callback(item:T, index:int, vector:Vector.<T>):Boolean
```

```
forEach(callback:Function, thisObject:Object = null):void
```

This method, too, can expect to iterate over objects of type T, and the callback takes the form:

```
function callback(item:T, index:int, vector:Vector.<T>):void
```

```
map(callback:Function, thisObject:Object = null):Vector.<T>
```

The map() method is identical in that the callback signature must be modified to take a T and return a T:

```
function callback(item:T, index:int, vector:Vector.<T>):T
```

Oddly, this prevents map() from being very useful, as the idea of map() is to process one type of information into another. If you are limited to converting type T to type T, you can no longer do very interesting mappings.

```
sort(compareFunction:Function):Vector.<T>
```

First, Vector has no sortOn() method. Second, this sort function, too, can operate on type T:

```
function compareFunction(x:T, y:T):Number
```

Finally, every method that nondestructively modifies a Vector by returning a new one returns a non-fixed-size Vector, even if the input Vector was fixed-size.

As you have seen in the examples, fast iteration — for..in and for each..in loops — works for Vectors just like it works for Arrays. Example 9-3 uses fast iteration and several of the methods of Vector to create a menu.

EXAMPLE 9-3   http://actionscriptbible.com/ch9/ex3

Vector Methods and Fast Iteration

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.text.TextField;
```

EXAMPLE 9-3 *(continued)*

```
import flash.text.TextFieldAutoSize;
import flash.text.TextFormat;

public class ch9ex3 extends Sprite {
  protected var labels:Vector.<TextField>;

  protected const DEFAULT_FORMAT:TextFormat
    = new TextFormat("_sans", 12, 0x000000, true);
  protected const SELECTED_FORMAT:TextFormat
    = new TextFormat("_sans", 12, 0x00ffff, true);
  protected const MARGIN:Number = 20;

  public function ch9ex3() {
    labels = new Vector.<TextField>;
    makeMenuItem("home");
    makeMenuItem("about");
    makeMenuItem("blog");
    makeMenuItem("contact");
  }

  protected function makeMenuItem(label:String):void {
    var tf:TextField = new TextField();
    tf.selectable = false;
    tf.defaultTextFormat = DEFAULT_FORMAT;
    tf.text = label;
    tf.width = tf.height = 0;
    tf.autoSize = TextFieldAutoSize.LEFT;
    if (labels.length > 0) {
      var lastTF:TextField = labels[labels.length-1];
      tf.x = lastTF.x + lastTF.textWidth + MARGIN;
    }
    addChild(tf);
    labels.push(tf);
    tf.addEventListener(MouseEvent.CLICK, onItemClick);
  }

  protected function onItemClick(event:MouseEvent):void {
    for each (var tf:TextField in labels) {
      tf.setTextFormat((tf == event.target)? SELECTED_FORMAT:DEFAULT_FORMAT);
    }
  }
}
}
```

Creating and Converting Vectors

To create a Vector, just use the Vector() constructor. The constructor takes two parameters: an initial length (which defaults to 0) and a determination of whether the Vector should be fixed-size (which defaults to false). Its signature is:

```
Vector(length:uint = 0, fixed:Boolean = false)
```

Vector literals

You can type `Vector` literals into your code just like `Array` literals. Simply use the `Vector` literal syntax:

```
var v:Vector.<int> = new <int>[1, 2, 3, 4, 5];
```

The literal syntax ends in the same way you'd write an `Array` directly into code, but it begins with the `new` keyword and the actual type used in the `Vector` instance. If this results in a compiler error, you may need to use a more recent compiler. Using `push()` with as many elements as you would like accomplishes the same thing with only a little more code, so you use `push()` to instantiate `Vectors` in the examples in this chapter for the sake of compatibility.

Version

FP10, compiler-dependent. Literals will compile in Flash Builder 4 and Flash Professional CS5 and above. Will not work in older builds of the Flex 4 SDK or Flash Professional CS4. ∎

Converting Types of Vectors

Remember that parameterized types are not compatible with each other, even if their actual types are related by inheritance. Consider the code:

```
var stringVector:Vector.<String> = new Vector.<String>();
var objectVector:Vector.<Object> = stringVector;
//if they were compatible, we could do this assignment
objectVector[0] = new Object();
var s:String = stringVector[0]; //uh oh, this isn't a String
```

By allowing `Vectors` to be compatible like their type parameters, you introduce a bidirectional equivalence that really only works one way. So parameterized types are always thought to be independent and incompatible.

However, ActionScript 3.0 does let you convert a `Vector` of one actual type to that of another actual type (see Example 9-4). By calling the top-level `Vector()` function (looks like casting), you can convert one `Vector` to another. All the elements will be converted to the destination type, and if this isn't possible, a runtime type error will be raised.

EXAMPLE 9-4 `http://actionscriptbible.com/ch9/ex4`

Collected Snippets: Converting Vectors

```
var numVector:Vector.<Number> = new Vector.<Number>;
numVector.push(1.414, 2.718, 3.142);
trace(numVector); //1.414,2.718,3.142
var intVector:Vector.<int>;
intVector = Vector.<int>(numVector);
trace(intVector); //1,2,3
```

This way, you can easily convert `Vector`s when their actual types are compatible. Particularly, this is helpful to run `toString()` on the contents of a `Vector` without using `map()` which, as you may recall, doesn't allow you to map to a different type.

Converting a Vector into an Array

The ease of converting `Vector` types might leave you optimistically trying the top-level `Array()` function to convert a `Vector` into an `Array`, perhaps so you can use it with some code written for an earlier version of Flash Player.

```
var array:Array = Array(intVector);
trace(array); //1,2,3 (looks good)
trace(array.length); //1 (wait a minute)
trace(getQualifiedClassName(array[0])); //_AS3_.vec::Vector.<int>
```

Alas, although it looks good at first glance, the `Array()` function actually inserts the `Vector` as the first index of the `Array`, rather than converting the elements of the `Vector` into elements of an `Array`. You actually have to do it by hand.

```
array = new Array();
for (var i:int = 0; i < intVector.length; i++) {
  array[i] = intVector[i];
}
trace(array); //1,2,3
trace(getQualifiedClassName(array[0])); //int
```

Similarly, you must convert `Array`s to `Vector`s by hand.

Summary

- `Vector` is an optimized version of `Array`.
- `Vector`s only store one type of element.
- `Vector`s have no gaps.
- `Vector`s can be fixed-size.
- When inserting, updating, and accessing elements, `Vector`s use the correct type rather than `*`.
- A `Vector` is a generic, or a type that can be used with any other type.
- When using an invocation of a generic, all the fields and methods in its interface have their type parameters (usually `T`) replaced by the actual type used.
- You can't create custom generics, but for most cases you should just use good object oriented programming to be abstract.
- Parameterized types are never compatible but may be converted manually if their types are compatible.

Objects and Dictionaries

A ll classes extend `Object` — it is the root of the ActionScript class hierarchy. Despite their inauspicious roots, `Objects` can be truly useful as a data structure. In Chapter 8, "Arrays," I showed how to use arrays to store associative data. In this chapter you will see how to use the `Object` and `Dictionary` data types to store and retrieve this kind of information and explore other situations where `Objects` are useful.

Working with Objects

The class `Object` is found at the root of the type hierarchy of all classes. Simple classes like `Number` and `String` extend `Object` directly; classes like `Sprite` descend from `Object`. In other words, every object in the ActionScript 3.0 world is an `Object`. The `Object` class by itself doesn't do much and would hardly merit its own chapter but for the interesting property that the `Object` class is one of the few classes that is *dynamic*.

Dynamic Classes

Dynamic classes can be extended at runtime with new properties and methods. This means you can take the data and operations that define an object and rename them, rewire them, or add to them while the program is running. Programming with dynamic classes is usually a poor choice. If you were programming a dinner set as a dynamic class, some code you don't control could overwrite your `saltShaker` property to dispense habanero peppers, ruining a perfectly good meal. A chaotic world like that is no world to live in, so every class I write in this book is going to be *sealed*, or closed for modification, at runtime. Without this guarantee, you can't follow the principles of object oriented design.

Note

Classes are sealed by default, but you can create your own dynamic (unsealed) class by using the keyword `dynamic` in front of your class definition:

```
public dynamic class UnsealedClass
```

On the other hand, combining this infinite expandability with an inherently empty class yields a perfect method of storage. This is how you come by `Object` as a data type. Why is `Object` "empty"? Because you need its subclasses to be able to do almost anything, `Object` itself does almost nothing.

You can think of `Object` as the clay from which everything else in the ActionScript world is crafted. When you write classes, you can sculpt a form in this clay and fire it, make it immutable, and ensure that all instances of the class have the same form. Or, you can choose to use the clay in its raw form, creating what you need for one time only, without writing a class. You can use `Object` for single-serving, throw-away structures.

Creating Objects

You can create a new `Object` with its constructor. The `Object()` constructor takes no parameters and makes an empty `Object`:

```
var o:Object = new Object();
```

`Object`, like `Number` and `String`, also has a literal form. You can create an `Object` from scratch with preset values simply by typing it as a literal. The literal form for `Objects` is a comma-separated list of name-colon-value pairs contained in curly braces:

```
var o:Object = {name: "Roger", hair: 0xff0000, hobbies: myHobbies};
```

The names of the properties don't need to be in quotes, and the values can be of any type. Values can also be references to other objects. In this example, you set the `hobbies` property of the `Object` to the `myHobbies` variable. Values can even be functions.

The names of a property in an object can be any string, including strings that aren't necessarily valid identifiers. This means you can store a property such as `99bottles` in an object, but you can't make a variable named `99bottles` because identifiers aren't allowed to start with a number.

Accessing Object Properties

Like properties of any class instance, you can access the properties of an object using either dot notation or bracket notation. The difference, besides syntax, is that you can use dot notation only with property names that are also valid identifiers. You can use these methods to both read from and write to objects, as shown in Example 10-1.

EXAMPLE 10-1 `http://actionscriptbible.com/ch10/ex1`

Reading from and Writing to Objects

```
var shape:Object = new Object();
shape.name = "hexagon";
shape.sides = 6;
shape["color"] = 0x00ff00;
```

```
shape["stroke weight"] = 2;
shape["describe"] = function():String {return "A " + this.name;};
trace(shape.describe() + " has " + shape.sides + " sides.");
// Displays: A hexagon has 6 sides.
```

This example demonstrates that function objects, like any other type of data, can be stored in an object instance. This is not a special case, but just another kind of object stored away. Typing `shape.describe` or `shape["describe"]` refers to the function object stored as `describe`, and typing the parentheses invokes that function.

You might notice that the bracket notation looks just like array access, and you can employ this "syntactic sugar" to use `Objects` as associative arrays.

toString()

There is one method defined on `Object` that you will find yourself using not just on `Objects` but on their subclasses. The `toString()` method is defined on `Object`, so it is available to all classes. It is used to generate a string representation of the instance. You can override this method in your own classes to provide a description of the instance and its contents for debugging and logging.

Note that when you use an object in a `String` context, Flash Player calls the `toString()` method on the object to convert it to a `String`. You can see `toString()` in action in Example 10-2 later in this chapter.

Using Objects and Dictionaries as Associative Arrays

An associative array is a data structure that stores items by index. It functions like a filing cabinet in that if you know the name of what you're looking for, you can find it by looking it up. The name, or the thing printed on the edge of the folders in the analogy, is called a *key*, and the item itself, or the folder you find, is called a *value*. Associative arrays, like a filing cabinet, let you add a new folder, find a folder by its name, remove a folder from the cabinet, look to see if a folder by a certain name exists, and flip through all the folders in the cabinet. The essential operation, however, is the lookup: retrieving values by name is what makes an associative array.

Associative arrays maintain a many-to-one relationship between keys and values. A key refers to the same value every time, an essential rule. However, you could easily have one value that is filed under multiple keys. For example, you might want to create a long, thin `Noodle` object and store it under "Angel Hair," but also under "Cappellini." Both names refer to the same kind of pasta, and both keys can refer to the same value: the instance of `Noodle`.

You might also hear associative arrays referred to as hashes, hashtables, maps, or dictionaries. These are all names for the same kind of data type. The terms "hashtable," "hash," and "map" all refer to a particular way to implement this data structure, in which a hash function is applied to the keys to evenly distribute the values in memory and to later look them up. All this is handled inside the ActionScript Virtual Machine; all you need to know is that an `Object` can be used as an associative array.

183

This kind of associative data is incredibly useful and is found all around us, in common programming situations, exposed to plain view on the internet, and in metaphors for the real world you can use to structure code. Most search engines, for example, expose associative data to you right in the address. Consider the URL www.google.com/search?q=hashtable&safe=active. After the question mark (?) is an associative array of keys and values. The query is stored as the key q, and its value is hashtable because you're searching for information on hashtables. The parameter for the search engine's content filtering feature is named safe, and its value is active because you don't want inappropriate results. Who knows what kind of lewd examples students in computer science courses might be posting right now? This example uses key-value pairs to represent parameters, as with any form on the internet that uses HTTP GET. It's one key organizational technique to refer to complex data by reference, which is just what associative arrays do.

In the example from the previous section, I demonstrated most of the necessary properties of an associative array using an Object. You stored and retrieved properties, or values, with dot and bracket notation. In this section, you see how to check for the existence of values, iterate through values, and remove values.

Comparing Arrays, Objects, and Dictionaries

Chapter 8 explained how to use an array to store associative arrays. I recommend that you use arrays only for numerically indexed data. Using an Array object creates the impression that the data can be accessed by numeric index and that useful Array methods such as indexOf() and splice() can be used. However, none of these methods apply to properties that are stored by key rather than index. You could create even more confusion by storing values in an Array instance both by index and by associative key. Or consider the capability to store values with keys such as 12. Using an array to access a property like that could confuse your code beyond use. Use Objects to store values by a textual key, and use Arrays to store values by numeric index.

Objects are ideal for storing key-value pairs. Using bracket notation with an Object instance looks just like looking up an array index by number, so you can think of it like an Array.

Dictionaries (or, I should say, Dictionarys) can also be used to store key-value pairs. They have an important enhancement, however.

Objects use strings as keys. This is sufficient for most purposes, but say you need a quick-and-dirty way to annotate class instances with certain additional information. Strings may not always be sufficient for this purpose. For example, you might be meeting a bunch of new people at a party.

```
var guy1:Person = new Person("Bob");
var guy2:Person = new Person("Charlie");
var girl1:Person = new Person("Alice");
```

You're trying to remember stuff about each person, so you use an Object to store information by the person's name.

```
var notes:Object = new Object();
notes["Bob"] = "Likes games";
notes["Charlie"] = "Super organized";
notes["Alice"] = "Enjoys drawing";
```

This works, until you meet a second person named Alice! Suddenly, there's no way to keep track of both Alices independently. Or heaven forbid, you might forget some of their names, but without a name you can't find anything.

```
var girl2:Person = new Person("Alice");
```

This would be a perfect opportunity to use a `Dictionary` object, shown in Example 10-2. `Dictionary` objects can use the objects themselves as keys. Instead of looking up a name, you can look up the entire person to get the data associated with that person.

EXAMPLE 10-2 `http://actionscriptbible.com/ch10/ex2`

Dictionary Objects

```
package {
    import com.actionscriptbible.Example;
    import flash.utils.Dictionary;
    public class ch10ex2 extends Example {
      public function ch10ex2() {
        var guy1:Person = new Person("Bob");
        var guy2:Person = new Person("Charlie");
        var girl1:Person = new Person("Alice");
        var girl2:Person = new Person("Alice");

        var notes:Dictionary = new Dictionary();
        notes[guy1] = "Likes games";
        notes[guy2] = "Super organized";
        notes[girl1] = "Enjoys drawing";
        notes[girl2] = "Plays piano";

        trace(girl1, notes[girl1]);
        trace(girl2, notes[girl2]);
      }
    }
}

class Person {
  public var name:String;
  public function Person(name:String) {
    this.name = name;
  }
  public function toString():String {
    return name;
  }
}
```

By using objects as keys, you can easily associate them with other data. I like to think of this like attaching sticky notes to something. Your `Dictionary` is a stack of sticky notes. When you attach a sticky note to your stapler, you don't actually modify the stapler, but you can always find that information if you have access to the original object — the stapler.

If you want to ensure that the objects you use as keys aren't kept around unnecessarily by a `Dictionary` object, you can pass `true` as the `weakKeys` parameter in the `Dictionary` constructor:

```
new Dictionary(true);
```

185

The value defaults to `false`. For more information, see the discussion on weak references in relation to event handling in Chapter 20, "Events and the Event Flow."

`Dictionary`s follow the same syntax for all the operations (inserting, deleting, setting, and retrieving) as an `Object`, so the techniques you learn in this chapter for using `Object`s also work with `Dictionary` instances.

Testing for Existence

You can test for the existence of a key in an associative array either by performing a lookup or using the `in` operator.

The `in` operator allows you to check on the existence of properties, methods, or values of an instance, as long as they are accessible. The operator returns a `Boolean` value. You can use the `in` operator to see if a key is associated with any value in an associative array:

```
var notes:Object = {Roger: "hereiam"};
"Roger" in notes; // Returns true
"Bailey" in notes; // Returns false
```

You can use this operator to investigate instances of classes for methods and public properties, which they support, as well, as shown in Example 10-3.

EXAMPLE 10-3 `http://actionscriptbible.com/ch10/ex3`

The in Operator

```
package {
  import com.actionscriptbible.Example;
  import flash.display.Sprite;
  public class ch10ex3 extends Example {
    public function ch10ex3() {
      var sprite:Sprite = new Sprite();
      sprite.alpha = 1;
      trace("alpha" in sprite); //true, that's a property
      sprite.getChildByName("subclip");
      trace("getChildByName" in sprite); //true, that's a method

      var person:Person = new Person("Ben");
      trace("name" in person); //true, Person has a name
      trace("SSN" in person); //false, because it's private!
      trace("unicorn" in person); //false, that's just silly
    }
  }
}
class Person {
  public var name:String;
  private var SSN:Number;
```

```
  public function Person(name:String) {
    this.name = name;
  }
}
```

This example illustrates finding methods and properties in both built-in and custom classes. It also shows that in respects the visibility of properties.

The old-school method of checking for existence by seeing if a lookup fails still works:

```
trace(notes["Josh"]); // Displays undefined
if (notes["Josh"]) {
  trace(notes["Josh"]);
} else {
  trace("I haven't met Josh yet.");
}
// Displays: I haven't met Josh yet.
```

Note

This method can be a little dangerous because several values can be coerced to false. If you say that Josh is a zero (notes["Josh"] = 0), for example, the else branch executes. Either compare your lookups directly to null or use the in operator to be safe. ∎

All Objects (meaning instances of all classes, since all classes inherit from Object) can provide this information with the hasOwnProperty() method, which returns true if the key you pass it is found as an instance variable, constant, method, or dynamically added property.

Removing Properties

You can use the delete operator to remove key-value pairs from an associative array. Apply the delete operator before the value (expressed the same way as you would look it up), and both the key and its associated value will be removed from the associative array:

```
var pastas:Object = {tortellini: 2, gemelli: 14, spaghetti: 9};
trace(pastas["spaghetti"]); //9
delete pastas["spaghetti"];
trace(pastas["spaghetti"]); //undefined
```

The delete operator also works on dynamically added properties and methods of instances of dynamic classes.

Note

The delete operator returns true if deletion succeeds. When using delete on a numerically indexed item in an Array, the length property is not updated. You cannot delete instance properties that are not dynamic, or local variables. To release objects, you can remove all references to them, such as by setting variables to null. ∎

Iterating

You can iterate through the key-value pairs of an associative array — whether it be an `Array`, `Object`, or `Dictionary` — with the `for..in` and `for each..in` loops introduced in Chapter 2, "ActionScript 3.0 Language Basics." Use the `for..in` loop when you need access to the keys, and use the `for each..in` loop when you are interested in the values only, as shown in Example 10-4.

EXAMPLE 10-4 `http://actionscriptbible.com/ch10/ex4`

Iterating Associative Arrays

```
package
{
  import com.actionscriptbible.Example;
  public class ch10ex4 extends Example {
    public function ch10ex4() {
      var characters:Object = {Roger: "The Author", Kumo: "The Cat"};

      trace("Roger" in characters); //true
      trace("Bailey" in characters); //false

      //iterate through the values
      for each (var value:String in characters) {
        trace("I see", value);
      }

      //iterate through the keys
      for (var key:String in characters) {
        trace(key, "is", characters[key]);
      }
    }
  }
}
```

Using Objects for Named Arguments

As you learned in Chapter 3, "Methods and Functions," ActionScript 3.0 gives you added flexibility when passing parameters to functions. You can leave out arguments with default values, and you can accept variable-length argument lists with the `...rest` parameter. Every so often it might be necessary for a function to accept more than a handful of possible arguments, in which case identifying the argument's intention from its position in the argument list alone can be a memory exercise.

For example, the following code sets a particular line style in preparation to draw some strokes on a `Sprite`:

```
var sprite:Sprite = new Sprite();
var g:Graphics = sprite.graphics;
```

```
g.lineStyle(1, 0xff0000, 1, true, "none", "round", "round", 20);
g.lineStyle(1, 0xff0000, 1, true, LineScaleMode.NONE, CapsStyle.ROUND,
  JointStyle.ROUND, 20);
```

The last two lines do the same thing. The function call is almost incomprehensible when you use plain strings for the three parameters near the end. When you use the static constants for these variables, the readability improves some.

In these cases, when the number of parameters to the function is not the result of a design flaw, you might wish you could use named arguments. Well, tough cookies, ActionScript 3.0 doesn't use named arguments, but you could accept an Object argument that stores multiple values which would otherwise be arguments. These values are accessible by named keys, almost like named arguments! Because you can't check the passed Object at compile time for correctness, it's best to only use an Object for named arguments that have defaults, so that if someone passes an Object in with a missing or invalid property, which isn't checked by the compiler, you can assign it the appropriate default value for that parameter.

Let's go back to the example and pretend instead that you rewrote the lineStyle() function to take an Object parameter. The call might look very different:

```
g.lineStyle({
  thickness: 1,
  color: 0xff0000,
  alpha: 1,
  pixelHinting: true,
  scaleMode: LineScaleMode.NONE,
  caps: CapsStyle.ROUND,
  joints: JointStyle.ROUND,
  miterLimit: 20
});
```

This use of a one-time, or *anonymous*, object to name parameters can increase readability of your code, but at the cost of compile-time argument checking. I recommend that you use this approach sparingly.

Using Objects as Nested Data

By inserting Objects inside Objects, you can create nested tree-like structures. By using dot notation, it is easy to traverse deeply nested trees of Objects.

```
plants.veggies.underground.carrot;
```

XML as Objects

In ActionScript 1.0 and ActionScript 2.0, it was not uncommon to convert XML structures into Object trees for convenience because traversing XML could be somewhat unwieldy. With E4X, however, traversing XML is just as easy. Also, E4X lets you handle structured data with much more elegance and sophistication than nested Objects. I recommend using XML for nested data. Chapter 11, "XML and E4X," shows how to work with XML in depth.

JSON

The syntax used to declare `Object` literals is simple and efficient enough for most uses that it has been adopted by many web programmers as a lightweight alternative to XML. For example, consider the following JavaScript `Object`:

```
var book =
{"title": "ActionScript 3.0 Bible 2nd Ed.",
 "ISBN": "978-0-470-52523-4",
 "author": "Roger Braunstein",
 "chapters": [
   {"title": "Objects and Dictionaries",
    "classes": ["Object", "Dictionary", "Array"]
   },
   {"title": "XML and E4X",
    "classes": ["XML"]
   }
 ],
 "rating": "awesome"
}
book.chapters[0].title; //"Objects and Dictionaries"
book.rating; //"awesome"
```

The `Object` assigned to `book` is a deeply nested data structure using only `Objects` and `Arrays`. Dot and bracket notation can be used to traverse it. Surround that object with quotes instead and it becomes a string. JavaScript interpreters simply have to `eval()` the string to turn the string into the object it represents. That is, in a nutshell, JSON.

Unfortunately, this trick doesn't work in ActionScript 3.0, because Flash Player does not compile or run ActionScript code at runtime, and `eval()` will throw an error if you ever call it. So E4X and lack of `eval()` interpretation are two good reasons to prefer XML for nested data.

If you must use JSON, however, Adobe provides a JSON parser in the corelib library available at `http://code.google.com/p/as3corelib/`.

Summary

- All classes extend `Object`.
- `Object` is an empty class that is dynamic.
- You can create and modify properties and methods of dynamic classes at runtime.
- Both `Object` and `Dictionary` implement associative arrays.
- Associative arrays store key-value pairs. In `Objects`, the keys must be strings. In `Dictionary`s, they can be any object.
- Anonymous objects can be used to pass named arguments.

XML and E4X

ActionScript 3.0 includes thorough, language-level support for XML. Not only can you create, load, and parse XML, you can write XML literals directly into ActionScript 3.0 code. A robust XML manipulation and query language, E4X, provides you with the tools to quickly and easily get the information you need. E4X, too, is implemented at the language level to make XML manipulation succinct.

Getting Started with XML in ActionScript

XML — the eXtensible Markup Language — is a format for storing any kind of hierarchical data. It has universal adoption and countless uses. XML is designed to be minimal; as the name *extensible* implies, you can design your own language that uses XML for its general structure but adds on more specific grammar.

XML References

This chapter assumes a basic knowledge of XML's structure and syntax. If you are unfamiliar with XML, you may want to check out the W3C Schools XML Tutorial (`http://www.w3schools.com/xml/`), the Wikipedia entry on XML (`http://en.wikipedia.org/wiki/XML`), or one of the many books available on XML. Let's agree on some terminology:

```
<?xml version="1.0" encoding="UTF-8"?>
<parentNode>
  <childNode attributeName="Attribute Value">
    Node Content
  </childNode>
```

```
      <childlessChildNode/>
      <!-- comment -->
   </parentNode>
```

The XML document begins with the *document declaration* and contains exactly one *root node* — `parentNode` here. Data is represented by *nodes*, or *elements* — `parentNode` and `childNode` — enclosed within start and end *tags*. Nodes can contain other nodes — like `parentNode` — or they can be empty — like `childlessChildNode`. Some elements contain *attributes* as name-value pairs within their start tags, which store additional information without nesting. Node names and attributes may be *namespaced* if the namespace is defined in the root node (not shown). Some nodes may contain *text nodes* — such as `Node Content` here — that are plaintext. Sometimes these text nodes use *character data*, or CDATA, sections to avoid escaping XML control characters in the text. XML may also contain *namespace declarations*, *comments*, or *processing instruction* nodes. An *element* is a node with a start and end tag — there are only three elements in this example — whereas the term *node* is more general and applies to text nodes, comments, and processing instructions like the document declaration.

E4X References

I can't talk about XML in ActionScript without talking about E4X. E4X is part of ActionScript's language specification. Besides defining the `XML` and `XMLList` classes that I'll soon cover, E4X provides operators and syntax you can use to write complex queries in a compact way and even defines how XML can be written in code. E4X is simply a blanket term for the pieces of ActionScript 3.0 that let you use XML data natively.

E4X, like much of the core ActionScript 3.0 language, is a standard. In fact, its name is shorthand for "ECMAScript for XML," defined in document ECMA-357. It's the same XML access standard that's used in JavaScript version 2.0.

You can read the official E4X spec at ECMA's site (`http://bit.ly/e4xspec`). Additionally, much of the information in this chapter is distilled in the article "AS3 E4X Rundown" (`http://dispatchevent.org/roger/as3-e4x-rundown/`).

XML Literals

`XML` literals, like `Number` and `String` literals, allow you to add XML directly into source code without having to load or parse it. Here's an example of an XML literal being assigned to a variable of type `XML`:

```
   var employeeList:XML =
     <employeeList>
       <employee>
         <name first="Conan" last="O'Brien" />
         <title>Host</title>
       </employee>
       <employee>
         <name first="Andy" last="Richter" />
         <title>Sidekick</title>
       </employee>
```

```
            <employee>
              <name first="Max" last="Weinberg" />
              <title>Band Leader</title>
            </employee>
          </employeeList>;
```

The preceding example shows XML data being assigned to a variable called `employeeList`. You can put the XML straight into your ActionScript file, as long as it's valid XML. Don't include the document declaration, but you can include XML comments, CDATA sections, and namespaces to your heart's content.

Note

AS3 lets you load XML from external sources. However, for simplicity's sake I use literals for examples in this chapter. To learn more about loading data from external sources, check out Chapter 27, "Networking Basics and Flash Player Security." ∎

You can use ActionScript expressions in XML literals by enclosing the expression in curly braces { }. The expression will be evaluated and its value substituted where the braces were. Now you can fill a data set with dynamically generated values, like so:

```
var squareLength:Number = 10;
var circleRadius:Number = 5;
var shapeData:XML =
  <shapeList>
    <shape type="square" size={squareLength} />
    <shape type="circle" size={circleRadius} />
  </shapeList>;
trace(shapeData.toXMLString());
```

The XML method `toXMLString()` returns its data as a string, but it retains the XML syntax and formatting. It won't necessarily respect the exact formatting of its input, because the method re-creates the `String` representation from the XML object data in memory. When run, this snippet prints the contents of `shapeData` as an XML string:

```
<shapeList>
  <shape type="square" area="10" />
  <shape type="circle" area="5" />
</shapeList>
```

As you can see, the two bracketed expressions from the XML literal were evaluated, and their values were placed directly into the XML. In addition to using inline expressions for XML attributes, you can use them for node names and text nodes, as Example 11-1 shows.

EXAMPLE 11-1   `http://actionscriptbible.com/ch11/ex1`

Inline Expressions in XML

```
package {
  import com.actionscriptbible.Example;
  public class ch11ex1 extends Example {
```

continued

EXAMPLE 11-1 *(continued)*

```
    override public function ch11ex1() {
      trace(makeNode("div", "i am a banana").toXMLString());
      //<div>i am a banana</div>
    }
    protected function makeNode(nodeName:String, contents:String = ""):XML {
      return <{nodeName}>{contents}</{nodeName}>
    }
  }
}
```

Note

I haven't properly covered `XMLList` objects yet, but you can create an `XMLList` literal by wrapping the nodes in an element with an empty node name. This is invalid XML, so it's used as a cue to E4X.

```
var colors:XMLList = <><red/><green/><blue/></>;
trace(colors.length()); //3
```

A Brief Introduction to E4X Operators and Syntax

An E4X expression usually combines methods of the `XML` and `XMLList` classes with operators. Many of the E4X operators are actually shortcuts for methods these classes define, which help keep complex searches readable. An E4X expression might look like this:

```
employeeList.employee[0].name.@first
```

These expressions, like others that use the dot operator, are evaluated from left to right. Every subexpression operates on the cumulative results thus far; in this example, every subexpression steps deeper into the XML structure to target a specific attribute of a specific node (`Conan`, from the earlier example). Drilling down is a common pattern for E4X expressions, but steps in your operation can also filter results or move up the XML tree.

You can think of the expression as "moving" through XML data, although it's more accurate to say that it generates intermediate data sets based on each subexpression. In this case, you start out with a whole XML data set, stored in the variable `employeeList`. You then retrieve all child nodes called `employee`, pick the first one, retrieve all its child nodes called `name`, and pick the attribute called `first` out of them.

In an expression like this one, some subexpressions can result in a single node, and some can result in multiple nodes. You can also write expressions to return special kinds of nodes like text nodes. E4X uses four classes to represent different kinds of XML data:

- `XML` — One piece of simple XML data; a single node. This is typically an element, but it can also store a text node, comment, or processing instruction. Defining all these seemingly different nodes as `XML` objects allows you to use the same methods and operations on all kinds of XML data.

- `XMLList` — A list of zero or more nodes. It may contain all elements, a single text node, or any number of nodes in any combination of node types. Each node in the list is an `XML` object. `XMLLists` support the same methods as `XML` objects, so they can be used interchangeably and transparently in E4X expressions. `XMLLists` are a bit different from `XML` objects in that they can contain multiple nodes, and these nodes can be from disparate places in an XML structure. Because of this, although they can be further filtered and queried, they cannot always be modified by adding or removing nodes. They are a set of possibly unrelated nodes, so Flash Player has no way of knowing where to apply your modifications. You can think of them as "result sets" and as read-only. `XMLLists` are indexed numerically like arrays, and they support fast enumeration with the `for each..in` loop.

- `Namespace` — Used not only to represent XML namespaces, but also ActionScript namespaces. Although you may use namespaced XML, you can do so without ever getting a handle on an actual `Namespace` instance. It's often easier to use the `namespace` and `use namespace` operators.

- `QName` — Represents a fully qualified name for an XML node or attribute. This includes both the name of the node or attribute and its namespace. E4X lets you write qualified names directly into code or with `Strings`, so it's rare that you'll end up using an actual `QName` instance.

All these classes are top-level: they exist in the default package, and there is no need to `import` them. Of the classes, you'll use `XML` and `XMLList` almost exclusively, especially if you work with simple XML that doesn't use namespaces.

In the space of one E4X expression, you can go back and forth between `XML` and `XMLList` seamlessly. For example, in the expression

```
employeeList.employee[0].name.@first
```

the subexpression `.employee` finds all child nodes named `employee` and returns a set, or `XMLList`. Then the subexpression `[0]` returns the first item in the set, always a single `XML` object (or `null`). As you learn about the different E4X operators and methods and see more examples of complex expressions, you'll get used to moving back and forth between `XML` and `XMLList` and see how they interact in more detail.

Note

Interestingly, neither of the `XML` or `XMLList` classes inherits from the other. However, they have most of the same methods and support the same E4X operations. It's a rare case when two classes are meant to be interchangeable but don't have some kind of formal type relationship. ■

Legacy XML Handling

In AS3, the top-level classes that I just enumerated handle XML using E4X; however, the older XML classes from previous versions of ActionScript are included for legacy support. The `XMLDocument` class, located in the `flash.xml` package, supports legacy XML handling. I won't cover legacy XML in this chapter. For more information, you can reference the `flash.xml` package in the AS3LR or reference any book or article on XML in ActionScript 2.0.

Querying XML

E4X makes it simple to query XML, extracting the information and structure you need from deep within an XML document. You can often query an XML node or set with either a method from `XML/XMLList` or a shorthand operator. I'll cover both the methods and their shortcuts, if applicable, simultaneously.

Many E4X operations can be said to correspond to certain *axes*. I've borrowed this term from other XML query languages such as XPath. If you think about a complex expression moving through an XML tree, each subexpression can move through the structure in a certain way. Each way of moving through the data is an axis. For example, the *parent* axis moves "up" to a node's immediate parents; the *descendant* axis includes all nodes "below" the node; and the *text* axis includes all text nodes contained by the node. The axis terminology is not used in much E4X literature I have seen, yet it's a good way to think and communicate about the different kinds of operations you can perform with a single E4X subexpression.

The Child Axis

The dot operator is the simplest tool in your E4X arsenal. It separates subexpressions, and, when used with a node name, moves down the child axis.

Example 11-2 defines the XML for a list of movies. I'll use this XML for the next several examples:

EXAMPLE 11-2 `http://actionscriptbible.com/ch11/ex2`

Collected Snippets: Querying XML

```
var movieList:XML =
  <list>
    <listName>My favorite movies</listName>
    <movie id="123">
      <title>Titus</title>
      <year>1999</year>
      <director>Julie Taymor</director>
    </movie>
    <movie id="456">
      <title>Rushmore</title>
      <year>1998</year>
      <director>Wes Anderson</director>
    </movie>
    <movie id="789">
      <title>Annie Hall</title>
      <year>1977</year>
      <director>Woody Allen</director>
    </movie>
  </list>;
```

Using the dot operator, you can access any of the `movieList`'s children. Simply write the node name of the child you want to access, as if it were a property of the `movieList` object. Notice here that you omit the root element, `<list>`, from the path:

```
trace(movieList.listName); //My favorite movies
```

The dot operator returns a set of nodes, matching all the children of the node or node set whose node name matches its argument. In other words, it converts an `XML` or `XMLList` into an `XMLList` because more than one child with the same node name may be present. If no matching nodes are found, it returns `null`.

Caution

There are some side effects to using the dot operator followed by a node name. You cannot use reserved words such as `class`, `var`, or `function` in subexpressions with the dot operator, even though these node names are perfectly legal XML. To access these nodes in the child axis, use the longhand `child()` method instead, which is not subject to these limitations because its argument is a `String`. On the other hand, interpreting the operand of the dot operator as a node name means that you can't access properties of `XML` instances. The property name would instead be read as a child's node name. So attributes of `XML` objects are made available as explicit accessors — methods — instead. The length of an `XMLList` named `xl` is accessed with `xl.length()`, not `xl.length`. Be wary of this; it's a common mistake. ∎

The corresponding methods for accessing the child axis are `child()` and `children()`. You'll see why there are two methods when we revisit `children()` in the next section. The `child()` method of `XML/XMLList` does the same thing as the dot operator: returns the node or node set's children that match its argument. You can pass the method a `String` or `QName` to specify the name of the child node(s) you're looking for, or an `int` if you are simply looking for the *n*th child node by index. Another way to get the example list's name is this:

```
trace(movieList.child("listName")); //My favorite movies
```

The Wildcard Operator

The wildcard operator (`*`) is used in several places in E4X to indicate that any value is acceptable for the corresponding argument. It is used where the argument is expected in both shorthand operators and method parameters. Although you can retrieve all the child nodes named `movie` by putting `movie` after the dot operator

```
trace(movieList.movie.length()); //3
```

you can get *every* child node by using a wildcard as the node name:

```
trace(movieList.*.length()); //4, includes <listName> node
```

Some axes have two associated methods: one that takes no parameters and returns all the XML objects in that axis, and one that returns only those that match the argument. This, too, is a convenience that allows the code to read a little better and lets you omit the argument:

```
trace(movieList.child("*").length()); //4
trace(movieList.children().length()); //4
```

Indexed Elements

For nodes with more than one of the same kind of child node, such as `movieList.movie`, and anywhere an expression results in an `XMLList`, you can get at individual XML elements in the set, determine how many elements there are, and determine if an element exists in the set.

The bracket operators (`[]`) let you access elements within an `XMLList` by index, just like an array. Remember that the first element will be at index 0.

```
trace(movieList.movie[1].title); //Rushmore
```

You can also use this syntax to write new values to the XML, just as you can append items to an array by assigning them to the next empty index.

```
movieList.movie[3]=<movie id="012"><title>Spaceballs</title></movie>;
```

Not that you needed another way to access children by name, but brackets also let you access child nodes by name, just like accessing named properties of `Objects`. This is another way to get at child nodes that have names illegal in or reserved by ActionScript. Steer clear of this and use the `child()` method instead.

Finding all elements in an axis and returning the *n*th result is equivalent to finding the *n*th element in that axis. Because you can pass an `int` to `child()` to find a child node by its order, these are equivalent:

```
trace(movieList.children()[0]); //My favorite movies
trace(movieList.child(0)); //My favorite movies
```

Right now, you might be getting the notion that there sure are a lot of different ways to do things in E4X. You're right. Just use whichever form makes the most sense to you.

The `length()` method of XML/XMLList returns the number of elements it contains. You've already used this one in a few examples. Yes, XML has a `length()` method; it returns 1, because all valid XML has a single root node.

The `childIndex()` method of XML returns the node's order in its parent node. You can use this to traverse around a node's siblings (an axis that has no E4X operation). So, to get to the next sibling of a node, you can look at the node whose index is one greater than its own:

```
var movie:XML = movieList.movie[0]; //get Titus node
movie = (movie.parent()).children()[movie.childIndex() + 1];
//get whatever's next
trace(movie.title); //Rushmore
```

This expression goes up (using the parent axis that I'll cover shortly) and then back down, using the index of the original node as a waypoint.

Finally, you can check whether a node is in a node set with the `contains()` method of XML/XMLList. (For an XML object, this compares the argument against the XML object.)

The Attribute Axis

To access attributes of a node or node set, you can use the *attribute identifier* operator (@). Use the operator followed by the attribute name that you want to find:

```
trace(movieList.movie[0].@id); // Displays: 123
```

In addition to reading attribute values, you can set them with this method:

```
movieList.movie[0].@id = 8675309;
```

If you want to access all the attributes for a tag, you can use the wildcard operator:

```
var movie:XML =
  <movie id="000" rating="*****">
    <title>The Big Lebowski</title>
    <year>1998</year>
    <director>The Coen Brothers</director>
  </movie>;
trace(movie.@*.toXMLString());
//000
//*****
```

Remember that any kind of XML element can be contained in an XML object; the result of querying the attribute axis is zero or more text nodes in an XMLList. When converted directly to a String, these run together like 000*****. When you use toXMLString(), each text node gets its own line in the output.

Similarly, when called on an XMLList, the attribute axis searches for the specified attributes on any node in the set.

```
trace(movieList.movie.@id); //123456789012
```

The attributes() and attribute() methods are provided to access all attributes, and attribute(s) by name, respectively.

```
movieList.movie[1].attribute("id"); // 456
movieList.movie[2].attributes(); // 789
```

The preceding code is functionally identical to the following code:

```
movieList.movie[1].@id; // 456
movieList.movie[2].@*; // 789
```

These attribute-axis methods are useful when you aren't sure if the node has an attribute with that value, or when an attribute name is reserved or illegal in ActionScript.

Tip

Other ways you can get to the attribute axis include these:

```
movieList.movie[1]["@id"]; //456
movieList.movie[1].@["id"]; //456
```

But just because you can doesn't mean you should. The more ways you use a single syntax, the less easily your code can be read. Use the @ symbol for attributes and the [] brackets for indexed children. ■

The Text Axis

If an element contains only a text node, toString() returns that text value automatically. Because the toString() method is called automatically when the expression is used in a String context, this provides a quick way to display the value of the text node.

However, some elements contain text as well as other child nodes, such as the following:

```
<div id="speech">
  Hear ye, hear ye!
  <hr/>
  All thine base doth belong to us!
</div >
```

To explicitly access the text node of an element, you can use the `text()` method, which returns an `XMLList` of text nodes:

```
trace(movieList.movie[1].title.text()); //Rushmore
```

There is no shorthand operator for the text axis.

The Descendant Axis

One powerful feature of E4X is the ability to directly access *descendant* nodes. A descendant is any node contained within a node, in any of that node's children, the children of those nodes, and so on. In other words, the descendant axis contains a node's children, grandchildren, great-grandchildren, great-great-grandchildren, and so on.

By using the *descendant accessor operator*, which is written as a double dot (`..`), you can make a deep dive to the data you want without worrying about what the path to that data might be. This works with elements but also with attributes and other types of XML objects. In the following example, you get all the movie title tags in one fell swoop:

```
trace(movieList..director);
```

This displays the following:

```
<director>Julie Taymor</director>
<director>Wes Anderson</director>
<director>Woody Allen</director>
```

See how simple that was? The descendant axis is even more valuable when it comes to larger XML trees. Let's try another example and pull every attribute from every node in the entire XML document:

```
trace(movieList..*.@*); //123456789012
```

The method `descendants()` is longhand for the descendant axis. Unlike other axes you've seen with two methods, one that accepts an argument and one that does not, this method does both, presumably because whether you choose all the descendants or just some, the result is less directly related to the original node than children. The parameter to `descendants()` is optional, so omitting it has the same effect as passing in a wildcard.

```
trace(movieList.descendants("year").text()); //199919981977
```

One thing you'll want to be aware of when using descendants is that all matches will be returned even if there are tags with the same name on different levels of the tree. The following example has a `<b>` tag, which is a descendant of another `<b>` tag:

```
var foo:XML = <a>
                  <b>
                    <c>
                        <b>foo</b>
                    </c>
                  </b>
              </a>;
trace(foo..b.toXMLString());
```

This displays the following:

```
<b>
    <c>
        <b>foo</b>
    </c>
</b>
<b>foo</b>
```

The descendant accessor can be a convenient shortcut, but be sure to structure your queries so that it won't pick up any unwanted extra nodes along the way.

The Parent Axis

If a descendant is a node or attribute contained within a particular element, an *ancestor* must be an element that contains the subject. Although there is no ancestor axis, there is a parent axis that contains the parent of the node when called on an XML instance, and the common parent of all the nodes in the set when called on an XMLList. If the nodes in the set have no parent, aren't normal elements, are the root element already, or don't have the same parent, undefined is returned. Because an element has only one parent (except the root element, which has none), this axis returns an XML object instead of an XMLList like most others do. The axis is accessed by the parent() method, which takes no arguments.

```
var title:XMLList = movieList.movie[1].title;
var director:XMLList = title.parent().director;
trace(title + " directed by " + director);
//Rushmore directed by Wes Anderson
```

There is no operator shorthand for the parent axis.

Custom Filter Axes

E4X facilitates powerful queries by allowing you to apply arbitrary tests to node sets within an expression. Elements that pass the test are returned. And, of course, you can continue processing these results further by chaining on additional E4X subexpressions.

To create your own filter axes, you simply write the test as a Boolean expression, using E4X with your ActionScript code as necessary. When placed in parentheses in an E4X expression, the expression is used as a filter. This may sound complicated when explained, but fortunately it looks natural in code:

```
var rushmore:XML = movieList.movie.(title.text() == "Rushmore")[0];
trace(rushmore.title + " directed by " + rushmore.director);
//Rushmore directed by Wes Anderson
```

Put in English, the E4X expression on the first line reads like this: "The first item, in the movie nodes of the list, **whose title node has a text node containing 'Rushmore.'**"

Let's look at a few more examples. The first filters by the movie's `year` node, and the second filters by the `id` attribute:

```
delete movieList.movie.(title == "Spaceballs")[0];
//get rid of this node for now since it doesn't contain a year or id.
var classics:XMLList = movieList.movie.(parseFloat(year) < 1990).title;
trace(classics); //Annie Hall
var idSort:XMLList = movieList.movie.(parseInt(@id) > 400).title;
trace(idSort);
//<title>Rushmore</title>
//<title>Annie Hall</title>
```

Before you filter on `id` and `year`, if it's still there you have to remove the "Spaceballs" movie you added in an earlier example. It's incomplete, and its lack of `year` node and `id` attribute will cause runtime errors if you use the shorthand E4X operators. However, if you use the corresponding methods instead of operators, you can avoid these errors.

Caution

When using filter axes with shorthand operators, all nodes that will be subjected to the test you write *must* have attributes and nodes that you reference in shorthand defined. To avoid this, use preceding expressions to narrow the input set to one that meets this requirement, or use E4X methods in your test expression, knowing that they may return `undefined`. ■

Another thing you have to keep in mind when using XML is that it's text based. All bits of XML, from node names to attribute values, are text data. An `@id` is a `String` even if it contains only numeric characters. Flash Player doesn't try to guess what type of data is being represented in an XML node. You have to tell it. This can be confusing, because comparison operators like `<` and `>` behave differently on strings and numbers. This can be a dastardly bug:

```
trace("9" > "10"); //true
trace( 9  >  10 ); //false
```

The results of these tests are so because the first character of the string "9" is greater alphabetically than the character "1", so the string is considered to have a higher value. This is why in the previous example I've used `parseInt()` and `parseFloat()` to ensure that the data is interpreted as, and compared as, the correct numeric types. Although you can take plenty of shortcuts in E4X — like using `year`, which is automatically converted into a `String`, instead of the more accurate `year.text().toString()` — a little explicitness goes a long way.

Caution

All XML elements are handled as text data. Ensure that you convert data to the proper types when necessary. ■

You can filter by any variety of criteria as long as you keep in mind that the filter terms are evaluated in the scope of the input node or node set. In this case, all the examples are searching for matches within the node set up to that point in the expression (`movieList.movie`). Here's an example that uses a string search on the director name:

```
var julies:XMLList = movieList.movie.(director.indexOf("Julie") != -1).title;
trace(julies); //Titus
```

Even though you look down into the `director` node within the expression, it's filtering on the `movie` nodes, and its output is a subset of `movie` nodes. This is why you can, after the filter expression, move down into the `title` child node.

Quick Reference

Let's review the different ways to query XML data in a chart. You can refer to this as a quick reference. The "node kind" axis is a special axis that gets information about its nodes. Learn more about it in the later section "Gathering Meta-Information about XML Nodes.

TABLE 11-1

E4X Query Operations

Axis	Description	Operator	Method(s)
child	children of the node	`.nodename`	`child(nodeNameOrIndex)` `children()`
descendant	children, grandchildren, etc.	`..nodename`	`descendants(optionalName)`
parent	node that contains this node		`parent()`
attributes	attributes of the node	`@attributeName`	`attribute(attributeName)` `attributes()`
filter	a custom filter	`(test expression)`	

node kind	child nodes that are a specific kind of node		`elements()`
			`text()`
			`comments()`
			`processingInstructions()`

Modifying XML

Of course, you are not limited to defining XML literals and querying them. E4X also defines operators and methods for modifying existing data by adding, removing, and updating values.

Inserting Nodes

For the following examples, let's update the movie list by adding the data for another movie.

```
var anotherMovie:XML =
  <movie id="222">
    <title>Tron</title>
    <year>1982</year>
    <director>Steven Lisberger</director>
  </movie>;
```

You'll look at some different ways to insert the XML you defined in the variable `anotherMovie` into the `movieList` XML structure at the correct place. In the real world, you'll need to update XML structures when you're collecting data on the fly, such as if you are maintaining a partial copy in memory of a large data structure, adding on nodes as they are retrieved from a data source.

It should be no surprise by now that E4X gives you different ways to insert XML data. The two main approaches are, as usual, by using operators and by using methods of `XML` and `XMLList`.

Inserting with E4X Operators

E4X provides two operators for concatenating XML data. These are the XML concatenation operator (+) and the XML concatenation assignment operator (+=). These two operators are just like those that are provided in `String` contexts for concatenating `String` data, and indeed they work in much the same way.

The simplest way to combine XML data is by using the + operator. This takes two operands, either of type `XML` or `XMLList`, and places them in order in an `XMLList`. Using this operator is quite natural:

```
var fruits:XMLList = <plum/> + <peach/> + <strawberry/>;
trace(fruits.toXMLString()); /*<plum/>
<peach/>
<strawberry/> */
```

Here I added three `XML` nodes to create an `XMLList` of length three.

The += operator adds the XML node or nodes on the right to the `XMLList` on the left and assigns that value back to the variable on the left. Again, because the result is multiple nodes, it must be an `XMLList`, so the left operand must be an `XMLList` variable.

```
var movies:XMLList = movieList.movie;
movies += anotherMovie;
trace(movies.title.text()); //TitusRushmoreAnnie HallTron
```

Here, I create a list of the movie nodes and add Tron to the list. Tracing out the titles of the movies in the list verifies that I've modified the variable `movies`. What I haven't done, however, is modify the original XML structure stored in `movieList`:

```
trace(movies.title.text()); //TitusRushmoreAnnie HallTron
//the XMLList has the new movie...
trace(movieList.movie.title.text()); //TitusRushmoreAnnie Hall
//but the original XML is unmodified
```

When I used the += operator, I assigned the new value to the `XMLList` called `movies`, which was a list of all the `<movie/>` nodes. The moment I created the `movies` variable, I lost the context of the query; `movies` doesn't know or care where its nodes came from. It's not a copy, however. The nodes themselves are references. Let's prove this:

```
movies[0].@newAttribute = "testing"; //modify a node in the list
trace(movieList.movie[0]); //<movie id="123" newAttribute="testing">...
//it changes the original node
```

Notice that I modified an attribute of the node from `movies`, but the changes also appear in `movieList` because both variables reference the same node.

If you use an E4X query expression on the left side of an XML assignment, Flash Player uses it as the context for the assignment, rather than modifying an intermediate node set. If you do this instead of using the intermediate variable `movies`, you can actually modify a targeted section of the overall XML with a single operator:

```
movieList.movie += anotherMovie;
//providing the context in the LHS of the assignment...
trace(movieList.movie.title.text()); //TitusRushmoreAnnie HallTron
//adds the node to the correct context, not just a temporary XMLList variable
```

By using the E4X expression `movieList.movie` as the recipient of the assignment, I've instructed Flash Player to append the new movie as a sibling of the `<movie/>` elements inside the original XML tree.

You can use XML assignment to add text nodes and attributes as well as elements:

```
var annieHall:XML = movieList.movie.(title.text() == "Annie Hall")[0];
annieHall.* += <genre/>;
annieHall.genre += "Comedy";
trace(annieHall);
//<movie id="789">
//   <title>Annie Hall</title>
//   <year>1977</year>
//   <director>Woody Allen</director>
//   <genre>Comedy</genre>
//</movie>
```

Above, I append a new `<genre/>` element to `annieHall`'s children and append a text node to it by concatenating a `String`. Note that when you use objects other than `XML` or `XMLList` instances as operands to XML assignment and concatenation, they are converted to `Strings` and treated as text nodes. This can be convenient when inserting data.

Remember that you can create new nodes and attributes by assigning them to not-yet-existing locations:

```
annieHall.actor += <actor>Diane Keaton</actor>;
annieHall.actor.@role = "Annie Hall";
```

Caution

When using E4X query expressions on the left side of an XML assignment, you need to stick to E4X operators. If you use `XML/XMLList` methods, Flash Player evaluates the left side first, rather than using it for context. Because the expression evaluates down to a literal constant, this is like assigning something to a number or string. Try to simplify your expressions, and use only E4X operators.

```
movieList.children() += <test/>; //X the LHS evaluates to a list
12 += 4; //X it's just like this, assigning to a constant
movieList.* += <test/>; //O using operators prevents evaluation
```

If necessary, you can use an intermediate `XML` variable. Sure, the intermediate variable doesn't have context, but if you add children to it, you've modified it for everyone:

```
annieHall.actor += <actor>Diane Keaton</actor>;
annieHall.actor.@role = "Annie Hall";
//we've modified an intermediate XML variable
trace(movieList.movie[2]);
//but this is the original Annie Hall node
//and changes to it are global:
//<movie id="789">
//  <title>Annie Hall</title>
//  <year>1977</year>
//  <director>Woody Allen</director>
//  <genre>Comedy</genre>
//  <actor role="Annie Hall">Diane Keaton</actor>
//</movie>
```

Inserting with E4X Methods

If you need more control over exactly where you want your values to be added to an XML object, or if you prefer to avoid E4X syntax that approaches the mystical, you can use methods of `XML` to insert nodes. These methods include

- `appendChild(child:Object):XML` — Appends the `child` node to the target node's children (making it the first child, if necessary). The following lines are identical:

  ```
  a.appendChild(b);
  a.* += b;
  ```

- `prependChild(child:Object):XML` — Prepends the `child` node to the target node's children. Note that the += operator always appends, so use this method if you need the child node to come first.

- `insertChildBefore(anchor:Object, toInsert:Object):*` — Inserts the `toInsert` node immediately before the `anchor` node, which is expected to be a child (immediate descendant) of the target node. You can specify the `anchor` node using E4X or by passing in a reference to an XML node. Returns `undefined` if the target node is not an element or the `anchor` is not found.

- `insertChildAfter(anchor:Object, toInsert:Object):*` — Inserts the `toInsert` node immediately after the `anchor` node. See `insertChildBefore()`.

  ```
  var textList:XML = <string/>;
  textList.appendChild("hello");
  textList.appendChild(" world");
  ```

```
trace(textList); //hello world
textList.insertChildAfter(textList.child(0), " there");
trace(textList); //hello there world
textList.insertChildBefore(textList.child(2), " beautiful");
trace(textList); //hello there beautiful world
trace(typeof textList.insertChildAfter("hello", "HEY!")); //undefined
```

None of these methods applies to `XMLList`. All the methods modify their targets and return the modified target, as long as no errors are encountered.

As you can see, these methods give you precise control over where you're inserting nodes in an XML structure.

Removing Nodes and Attributes

Unlike inserting XML, there are no methods for deleting XML nodes. Instead, you simply use the `delete` operator. This removes a specified node from the tree. Let's say (hypothetically, of course) that I've grown tired of Woody Allen's neurotic musings and want him taken off my list. Running the following code

```
delete movieList.movie[2];
trace(movieList.movie);
```

displays

```
<movie id="123">
  <title>Titus</title>
  <year>1999</year>
  <director>Julie Taymor</director>
</movie>
<movie id="456">
  <title>Rushmore</title>
  <year>1998</year>
  <director>Wes Anderson</director>
</movie>
```

You can use the `delete` operator with E4X expressions to delete combinations of nodes at once. You can simplify the structure of the movie by removing all information but the title:

```
delete movieList.movie.director;
delete movieList.movie.year;
trace(movieList);
//<list>
//  <listName>My favorite movies</listName>
//  <movie id="123">
//    <title>Titus</title>
//  </movie>
//  <movie id="456">
//    <title>Rushmore</title>
//  </movie>
//  <movie id="789">
//    <title>Annie Hall</title>
```

```
//  </movie>
//</list>
```

The `delete` operator works for attributes as well as nodes. In the following example, I remove all attributes from the movie *Tron*.

```
delete anotherMovie.@*;
trace(anotherMovie); //<movie>... the id is gone.
anotherMovie.@id = 222; //let's add the id attribute back.
```

Duplicating XML

As I discussed in previous chapters, only base types are passed by value, and all complex objects in ActionScript are passed by reference. XML is no exception to this rule. Setting a variable to another XML object creates a reference to the original object, and any changes made to either variable are reflected across both variables. I've relied on this fact in many prior examples. But sometimes you actually want a copy:

```
var template:XML = <person><name><first /><last /></name></person>;
var me:XML = template;
me.name.first = "Roger";
me.name.last = "Braunstein";
var someoneElse:XML = template;
trace(someoneElse); //oops, we've modified the template instead:
//<person>
//  <name>
//    <first>Roger</first>
//    <last>Braunstein</last>
//  </name>
//</person>
```

You can clone XML data effortlessly by using the `copy()` method:

```
var template:XML = <person><name><first /><last /></name></person>;
var me:XML = template.copy();
me.name.first = "Roger";
me.name.last = "Braunstein";
var mario:XML = template.copy();
mario.name.first = "Mario";
mario.name.last = "Bros";
trace(template); //all good, keep the copies coming!
//<person>
//  <name>
//    <first/>
//    <last/>
//  </name>
//</person>
```

Replacing Nodes

The child nodes of an XML object can be replaced all at once by using the `setChildren()` method. This replaces all the child nodes with the XML that you provide. In the following example, you change the movie Tron (which has the ID 222) first to an empty element, and then to The Science of Sleep:

```
movieList.movie.(@id == 222).setChildren(null);
trace(movieList.movie.(@id == 222).toXMLString());
// Displays: <movie id="222">null</movie>
movieList.movie.(@id == 222).setChildren
(<title>The Science of Sleep</title> +
                                <year>2006</year> +
                                <director>Michel Gondry
                                     </director> +
                                <genre>Romance</genre>);
trace(movieList.movie.(@id == 222).toXMLString());
// Displays: <movie id="222">
//              <title>The Science of Sleep</title>
//              <year>2006</year>
//              <director>Michel Gondry</director>
//              <genre>Romance</genre>
//           </movie>
```

A similar method, called `replace()`, allows you to replace a single node with a new XML object. The new XML object can be anything and doesn't need to use the same tag name as the element being replaced:

```
movieList.movie.(@id == 222).replace("genre",
<category>Independent</category>);
trace(movieList.movie.(@id == 222).toXMLString());
// Displays: <movie id="222">
//              <title>The Science of Sleep</title>
//              <year>2006</year>
//              <director>Michel Gondry</director>
//              <category>Independent</category>
//           </movie>
```

For more information on the `setChildren()` and `replace()` methods, check the AS3LR.

Converting to and from Strings

XML objects are essentially text data. When interpreted as XML, they inherit the tree structure you're familiar with. Thankfully, it's easy to move back and forth between XML and strings. Besides the familiar `toString()` method, there are a few more options specifically for `XML` and `XMLList` objects.

Converting Strings to XML

You may find it necessary to convert string values to XML when working with data from a string-based source or an external text file. To do this, you can simply use the cast-like conversion function XML(). This looks just like casting the argument to XML. You should be careful to use only well-formed XML text when casting. Failure to do so will result in a runtime error.

```
var dialog:String = "Lorem ipsum dolor sit amet";
var xmlString:String = "<dialog>" + dialog + "</dialog>";
var xml:XML = XML(xmlString);
trace(xml.text()); //Lorem ipsum dolor sit amet
```

The XML() constructor also accepts any object that can be converted into XML, so you can accomplish the same conversion with this:

```
xml = new XML(xmlString);
```

Converting XML to Strings

As you've seen so far throughout the chapter, you can use the traditional toString() method with XML objects to display their contents as text. Flash Player decides how to convert the XML depending on the node type and contents. However, XML/XMLList also provides toXMLString(), which always represents the node or nodes with proper XML structure. This method is guaranteed to produce valid XML fragments. You can expect the output to be structurally and functionally equivalent to its input, but because it's re-created from the XML structure in memory, it's not always character-for-character identical. For instance, toXMLString() uses its own formatting rules to generate the proper whitespace; no spaces are included at the end of childless tags like <spaces />; nicknames for namespaces may be lost. Let's demonstrate different string conversion methods in the following example:

```
var meal:XML =
  <meal>
    <name>Dinner</name>
    <course number="1">
        <item>Salad</item>
    </course>
    <course number="2">
        <item>Potatoes</item>
        <item temperature="Medium Rare">Steak</item>
    </course>
 </meal>;
trace(meal.name); //Dinner
trace(meal.course[0]); //<course number="1">
                       //  <item>Salad</item>
                       //</course>
trace(meal.course[1].item[0].toXMLString()); //<item>Potatoes</items>
trace(meal.course[1].item[1].toString()); //Steak
trace(meal.course[1].item[1].@temperature.toXMLString());//Medium Rare
trace(meal..item.toString());
//<item>Salad</item>
//<item>Potatoes</item>
//<item temperature="Medium Rare">Steak</item>
```

Printing Pretty

There are a few options when it comes to formatting XML strings created by `toString()` and `toXMLString()`. The first of these is the `prettyPrinting` flag. Setting and retrieving XML option flags is covered more fully in the later section "Setting XML Options," but I'll introduce a few of them now. For now, I'll set XML options through their static properties on the XML class.

When the `prettyPrinting` flag is set, XML strings are produced with line breaks and indentation in the canonical manner, as you've used throughout the chapter. Setting this flag affects *all* XML objects; it is set to `true` by default.

```
var example:XML = <stooges>
                     <moe /><curly><larry>
    </larry>
                              </curly>
            </stooges>;
XML.prettyPrinting = false;
trace(example);
//<stooges><moe/><curly><larry/></curly></stooges>

XML.prettyPrinting = true;
trace(example);
//<stooges>
//  <moe/>
//  <curly>
//    <larry/>
//  </curly>
//</stooges>
```

Notice how some cleanup occurs automatically, removing most of the whitespace. This is actually an effect of another flag, `ignoreWhitespace`.

Setting the Number of Spaces Per Indentation

When you use the `prettyPrinting` feature, your text is automatically indented. The amount of indentation is specified by the XML option `prettyIndent`. This value holds the number of spaces used for each level of indentation as an `int`. The default value is 2, and it is ignored when the `prettyPrinting` flag is set to `false`. The following code

```
XML.prettyIndent = 0;
trace(example);
XML.prettyIndent = 8;
trace(example);
```

displays the text with 0 spaces per tab:

```
<stooges>
<moe/>
<curly>
<larry/>
</curly>
</stooges>
```

and then with 8 spaces per tab:

```
<stooges>
        <moe/>
        <curly>
                <larry/>
        </curly>
</stooges>
```

Normalizing Text Nodes

XML objects support multiple adjacent text nodes per element although, most times, only one text node is used per element. In the rare case that you might have more than one text node in a given element, you can use the `normalize()` method to clean up whitespace and combine these text nodes into a single, contiguous node. In the following example, you add a fourth Stooge to the list using a series of `appendChild()` calls.

```
//add "shemp" using 3 appendChild() calls
example.appendChild("sh");
example.appendChild("");
example.appendChild("emp");
trace(example.text().length()); //3
trace(example.text().toXMLString());
//sh
//
//emp
```

Notice that the length of the `text() XMLList` is 3, and three lines are printed when displaying this text. Normalizing will merge these three text nodes into one.

```
example.normalize();
trace(example.text().length()); //1
trace(example.text().toXMLString()); //shemp
```

Typically, nodes are normalized during parsing and many E4X operations. Only when multiple text nodes are intentionally added, as in the preceding example, will the `normalize()` method be used.

Loading XML Data from External Sources

The subject of loading data from servers is more involved than can be covered in a chapter on XML. I give loading remote data its due in Part VI, "External Data," but here's a teaser that should get you loading and parsing XML files from a web server in short order.

Loading XML in ActionScript 3.0 is just like loading any text, as shown in Example 11-3. You use a `URLLoader` with a `URLRequest` to fetch the text, and when it's loaded, you convert the text to XML, as described in the earlier section "Converting XML to strings."

EXAMPLE 11-3 `http://actionscriptbible.com/ch11/ex3`

Loading XML

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  import flash.net.URLLoader;
  import flash.net.URLRequest;

  public class ch11ex3 extends Example {
    public function ch11ex3() {
      var url:String = "http://actionscriptbible.com/files/hand.xml";
      var loader:URLLoader = new URLLoader(new URLRequest(url));
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
    }

    protected function onLoadComplete(event:Event):void {
      var loader:URLLoader = URLLoader(event.target);
      var loadedXML:XML = XML(loader.data);
      trace(loadedXML.toXMLString());
    }
  }
}
```

You can replace the URL in this example with the location of your data. When the loader is done loading, the `onLoadComplete()` method is called, which converts the data loaded by the loader into XML. From this point, you can use `loadedXML` the same way you've used any other XML data in this chapter.

Gathering Meta-Information about XML Nodes

XML instances let you inspect meta-information about the XML data they contain. This can be useful when automating the processing of XML objects. You can use methods to find the type of node and the type of content a node contains: either simple or complex.

Finding Node Types

To determine the type of node that you're working with, you can use the `nodeKind()` method on the node that you want to inspect. The function returns a `String` that describes the type of node. The string returned will be one of the following:

- `element`
- `attribute`

- `text`
- `comment`
- `processing-instruction`

Tip

To work with comments and processing instructions, you must disable the `ignoreComments` and `ignoreProcessingInstructions` flags, respectively, as they both default to `true`. See the later section titled "Setting XML Options." ■

The following example queries meta-information from several nodes from an XHTML document with inline PHP source:

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;
var xhtml:XML =
  <html>
    <head />
    <body id="main">
      Welcome!
      <!-- Start PHP -->
      <?php echo("hello world") ?>
      <!-- End PHP -->
    </body>
  </html>;
var node:XML;
trace(xhtml.body.nodeKind()); //element
trace(xhtml.body.@id.nodeKind()); //attribute

node = xhtml.body.text()[0];
trace(node.nodeKind(), node.toXMLString()); //text Welcome!

node = xhtml.body.comments()[0];
trace(node.nodeKind(), node.toXMLString()); //comment <!-- Start PHP -->

node = xhtml.body.processingInstructions()[0]
trace(node.nodeKind(), node.toXMLString());
//processing-instruction <?php echo("hello world") ?>
```

Determining the Type of Content in a Node

Nodes that contain only a single text node or no content at all are said to have *simple content*. Nodes that contain children are said to have *complex content*. To determine whether a node contains simple or complex content, you can use either `hasComplexContent()` or `hasSimpleContent()`. Either will do — just use whichever one makes more sense to you.

```
var contentTest:XML =
<test>
    <a />
    <b>Lorem Ipsum</b>
    <c>
```

```
      <d />
    </c>
  </test>;
trace(contentTest.a.hasSimpleContent());  //true
trace(contentTest.b.hasSimpleContent());  //true
trace(contentTest.c.hasSimpleContent());  //false
trace(contentTest.d.hasComplexContent()); //false
```

Using Namespaces

Namespaces are used in XML to group XML elements and attributes into cohesive units, in the same way that you use packages and custom namespaces to group classes into sets that work together. In XML, namespaces are defined and named on the root node. XML namespaces are identified uniquely by their URI (as opposed to ActionScript namespaces, for which a URI is optional). You can define a *default namespace* by declaring a namespace with no name.

```
<?xml version="1.0" encoding="UTF-8"?>
<rootNode
  xmlns:asb="http://actionscriptbible.com/ns/example"
  xmlns="http://actionscriptbible.com/ns/default">
</rootNode>
```

In the previous example, two namespaces are declared: a default namespace and `asb`.

Note
The URI for an XML namespace doesn't actually require anything to be located there; its purpose is to provide a unique string. Generally, URIs are thought to be one-of-a-kind, but so is the phrase "Grund staggle ipp brid hunf," which would work just as well as far as ActionScript is concerned. ∎

To specify an element or attribute as a member of a particular namespace, use the namespace name followed by a colon and then the node name or attribute name as usual:

```
<?xml version="1.0" encoding="UTF-8"?>
<rootNode xmlns:asb="http://actionscriptbible.com/ns/example">
    <asb:chapter>Working with XML</asb:chapter>
</rootNode>
```

When a default namespace is used, nodes and attributes are scoped to the default namespace when no namespace is specified:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Test</title>
  </head>
  <body>Hello namespaces!</body>
</html>
```

In the previous example, every element belongs in the XHTML namespace with URI `http://www.w3.org/1999/xhtml`.

Caution

Properly structured XHTML documents have a default namespace, as in the previous example. When using XHTML and other XML with default namespaces, it's easy to forget that every node is namespaced. If your E4X queries keep returning `null`**, you may have forgotten to open or use the proper namespace. You can find more on this common error at** `http://dispatchevent.org/roger/using-e4x-with-xhtml-watch-your-namespaces.` ■

I just covered the absolute basics of XML namespaces. If you want to know more, I recommend you read the W3C spec on namespaces or the concise and useful tutorial at `http://www.w3schools.com/xml/xml_namespaces.asp`.

Creating XML Namespaces in ActionScript

There are several ways to declare and use XML namespaces with E4X. For this section, let's consider parsing an MXML file for a Flex 4 application, which declares several namespaces. Flex uses namespaces to identify different libraries within the framework; this one simple example uses three Flex namespaces.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo">
    <s:Button label="Test" id="myButton"/>
    <mx:Slider direction="horizontal" id="mySlider"/>
    <fx:Script><![CDATA[
        protected const AUTHOR:String = "roger";
    ]]></fx:Script>
</s:Application>
```

Using the Namespace Class

First let's look at the `Namespace` class. Instances of `Namespace` represent namespaces, which can be used for scoping XML as well as ActionScript. These contain just one datum, the *namespace URI*. This is all it takes to identify a namespace in XML: the prefix you define in the root node is merely a handle for the namespace, just as the name of a `Namespace` variable is the handle to the actual namespace. In fact, your `Namespace` instance name and the prefix used in the XML don't need to match, although it certainly helps to avoid confusion.

If you want to access the button called `myButton` in E4X, you have to create a namespace handle for the Spark namespace in which it's contained, `library://ns.adobe.com/flex/spark`. One way you can do this is with a `Namespace` variable. You can pass the `Namespace` constructor a URI, or a prefix and a URI. (It's a bit of an odd constructor in this way.) The prefix is optional.

```
var s:Namespace = new Namespace("library://ns.adobe.com/flex/spark");
var s:Namespace = new Namespace("s", "library://ns.adobe.com/flex/
spark");
```

Both of these declarations are fine. Additionally, the prefix and URI of a `Namespace` instance can be read and written at any time using its `prefix` and `uri` properties. `Namespace` objects return their URI value by default when the `toString()` method is used.

Using the Namespace Keyword

Alternatively, you can use ActionScript's `namespace` keyword to declare namespaces. Namespaces declared with the `namespace` keyword are no different from those created with `Namespace` instances.

```
namespace s = "library://ns.adobe.com/flex/spark";
```

Think of classes in ActionScript. You declare them with the `class` keyword, but you can get references to classes as instances of `Class`. Likewise, you can get references to namespaces declared with the `namespace` keyword — such as by using `getDefinitionByName()`, which returns a `Namespace` instance when you pass it the name of a namespace.

The advantage of the `namespace` keyword is that it can be used with the `use namespace` keyword, which is covered in the later section "Querying namespaced XML nodes." Overall, the two methods differ only slightly. The `Namespace` class is more useful for throw-away namespaces you'll use infrequently. The `namespace` keyword is better for reusable namespaces. It can also be used for ActionScript namespaces.

Making Namespaces Available

As you learned in Chapter 4, "Object Oriented Programming," namespaces — like variables, constants, classes, and functions — have a visibility determined by where you place them and what visibility modifiers you use on them. If you are going to reuse a namespace across multiple classes, which is certainly the use case for ActionScript namespaces and may be useful for XML namespaces, you might want to make it public to any code that chooses to import it.

Say you want to reuse the namespaces `s` and `mx`. You can do so using either approach, as long as the namespace is the only public item in the package block. So let's create two files. Try the `namespace` keyword by including this text in the file `com/actionscriptbible/xmlns/s.as`:

```
package com.actionscriptbible.xmlns {
  public namespace s = "library://ns.adobe.com/flex/spark";
}
```

Try using a `Namespace` instance by including this text in the file `com/actionscriptbible/xmlns/mx.as`:

```
package com.actionscriptbible.xmlns {
public const mx:Namespace = new Namespace("library://ns.adobe.com/
flex/halo");
}
```

Either approach works for making an XML namespace reusable.

Querying Namespaced XML Nodes

Once you have a namespace with the same URI as a namespace in your XML document, you can use it to query namespaced nodes in XML. Whether the namespace is declared with the `namespace` keyword or the `Namespace` class, whether it is declared in the same block or imported from another package, as long as it's in scope, you can use it with E4X.

The rules for using XML namespaces are really no different from those for using ActionScript namespaces. I'll review these and cover one additional operator that is specific to XML namespaces.

Opening Namespaces

When you query for a node or attribute in E4X, it will be returned if it can be found in the namespaces that are currently open. However, no namespaces are open by default, so E4X queries only match nodes and attributes not in a namespace. You can add to the list of currently open namespaces with the `use namespace` keyword with the name of the namespace, as Example 11-4 shows. This statement, when placed inside a package block, opens the namespace for all code inside the block.

EXAMPLE 11-4   `http://actionscriptbible.com/ch11/ex4`

XML Namespaces in ActionScript

```
package {
  import com.actionscriptbible.Example;
  import com.actionscriptbible.xmlns.mx;
  import com.actionscriptbible.xmlns.s;

  use namespace s;
  use namespace mx;

  public class ch11ex4 extends Example {
    private const MXML:XML =
      <s:Application
        xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/halo">
        <s:Button label="Test" id="myButton"/>
        <mx:Slider direction="horizontal" id="mySlider"/>
        <fx:Script><![CDATA[
            protected const AUTHOR:String = "roger";
        ]]></fx:Script>
      </s:Application>

    public function ch11ex4() {
      trace(MXML.Slider.toXMLString()); //(no output)
      trace(MXML.Button.toXMLString()); //<s:Button...
    }
  }
}
```

In this example, you attempt to open the `mx` and `s` namespaces. However, only opening the `s` namespace works correctly, as it was declared as a `namespace` rather than a `Namespace` instance. Note that you still have to import the namespaces to reference them in the `use namespace` statement.

When a namespace is open, you can access nodes in that namespace without specifically referencing it. This is the same procedure for accessing items (functions, classes, methods, and so on) from an ActionScript namespace without specifying the namespace.

Opening a namespace is especially useful when you need to repeatedly access nodes and attributes declared in the namespace. Once opened, you can query nodes in the namespace to your heart's desire without ever resolving the namespace.

Using the Scope Resolution Operator

To explicitly specify in which namespace a node or attribute is, you can use the scope resolution operator (::). This operator has an identical use in ActionScript for specifying the namespace a class or method exists in. This operator can be used with namespaces declared with the namespace keyword as well as by creating Namespace instances.

```
package {
  import com.actionscriptbible.Example;
  import com.actionscriptbible.xmlns.mx;
  import com.actionscriptbible.xmlns.s;

  public class ch11ex4 extends Example {
    private const MXML:XML =
      <s:Application
        xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/halo">
        <s:Button label="Test" id="myButton"/>
        <mx:Slider direction="horizontal" id="mySlider"/>
        <fx:Script><![CDATA[
            protected const AUTHOR:String = "roger";
        ]]></fx:Script>
      </s:Application>

    public function ch11ex4() {
      trace(MXML.mx::Slider.toXMLString()); //<mx:Slider...
      trace(MXML.s::Button.toXMLString()); //<s::Button...
      var fx:Namespace = new Namespace("http://ns.adobe.com/mxml/2009");
      trace(MXML.fx::*.toXMLString()); //<fx:Script...
      trace(MXML.*::Button.toXMLString()); //<mx::Button...
    }
  }
}
```

In this example, you use the scope resolution operator to specifically look for Slider nodes declared in the s namespace and Button nodes declared in the mx namespace. You also create a local Namespace instance fx and look for any child nodes in that namespace. Finally, you can even use a wildcard with the scope resolution operator. Here you use it to look for a node named Button in *any* namespace. Because this doesn't require you to declare a matching namespace, you can use it for quick-and-dirty operations with namespaced XML.

The scope resolution operator also works for namespaced attributes, as shown in Example 11-5. As before, simply specify the namespace before the attribute name. The @ operator still comes first.

EXAMPLE 11-5 `http://actionscriptbible.com/ch11/ex5`

Scope Resolution Operator

```
package {
  import com.actionscriptbible.Example;
  public class ch11ex5 extends Example {
    public function ch11ex5() {
      var xml:XML = <root xmlns:myns="http://example.com/some/namespace">
        <node id="1" myns:color="red"/>
      </root>;

      var myns:Namespace = new Namespace("http://example.com/some/namespace");
      trace(xml.node.@id);
      trace(xml.node.@myns::color); //right
      trace(xml.node.@*::color); //right
      trace(xml.node.attribute("myns::color")); //wrong
      trace(xml.node.attribute(new QName(myns, "color"))); //right
    }
  }
}
```

Earlier you created a node with one namespaced attribute and accessed it through various methods. Again, the wildcard operator may be used to specify any namespace. If you're using E4X methods with namespaced nodes and attributes, you'll need to look to the `QName` class, which simply bundles an identifier with a namespace.

Setting the Default XML Namespace

Opening a namespace is one way to quickly be able to access multiple nodes in the same namespace without using `::` to specify the namespace for every node. However, there are still some drawbacks. If there are two nodes with the same name from different namespaces, you'll still need to specify which one you mean. More importantly, you'll still need to declare the namespace on any nodes you create.

If you're using only one namespace and this is the default one, or if you are going to be creating lots of nodes in the same namespace, you might want to set the default XML namespace. This is done with a unique E4X statement, `default xml namespace`. When you assign a value to this special variable, all code in the same scope (using the same scoping rules as ActionScript variables) uses the namespace as the default namespace. This has several effects, as shown in Example 11-6.

EXAMPLE 11-6 `http://actionscriptbible.com/ch11/ex6`

Default XML Namespace

```
package {
  import com.actionscriptbible.Example;
  public class ch11ex6 extends Example {
    private const MXML:XML =
```

```
      <s:Application
         xmlns:fx="http://ns.adobe.com/mxml/2009"
         xmlns:s="library://ns.adobe.com/flex/spark"
         xmlns:mx="library://ns.adobe.com/flex/halo">
         <s:Button label="Test" id="myButton"/>
         <mx:Slider direction="horizontal" id="mySlider"/>
         <fx:Script><![CDATA[
             protected const AUTHOR:String = "roger";
         ]]></fx:Script>
      </s:Application>

   public function ch11ex6() {
      var s:Namespace = new Namespace("library://ns.adobe.com/flex/spark");
      default xml namespace = s;

      var newXML:XML = <Application><Button label="new button"/></Application>;
      trace(newXML.toXMLString());
      //<Application xmlns="library://ns.adobe.com/flex/spark"...
      //any new XML you create uses this as the default namespace

      trace(MXML.Button.toXMLString());
      //the s namespace is open as the default namespace, so no need to scope

      MXML.appendChild(<VGroup/>);
      MXML.VGroup.appendChild(<CheckBox label="new control!"/>);
      trace(MXML.toXMLString());
      //these new nodes get assigned to the s namespace too.
      //...<s:VGroup>
      //     <s:CheckBox label="new control!"/>
      //   </s:VGroup>
   }
  }
 }
```

In this example, the default namespace is applied to new nodes you create in existing documents, new XML fragments, and any queries in which you don't otherwise specify the namespace.

Use a default XML namespace when you're querying or creating XML documents, which are primarily in a single namespace, such as an XHTML document.

Querying XML for Namespaces

One E4X axis we haven't talked about yet is the namespace axis. This, and its associated E4X method `namespace()`, retrieves XML namespaces from nodes. You can use this method in two ways. When passed no argument, it reports the namespace of the subject node:

```
      var buttonNS:Namespace = MXML.*::Button.namespace();
      trace(buttonNS.prefix, buttonNS.uri); //s library://ns.adobe.com/flex/spark
      trace(MXML.buttonNS::Button.toXMLString()); //<s:Button...
```

221

In this example, we cleverly utilize `namespace()` to retrieve the namespace of a node whose name we know but namespace we don't. Once we have the namespace reference, we can reuse it on other nodes that we know belong to the same namespace, or we can use it to get back to the original node.

The second way you can use `namespace()` is by passing it a namespace prefix. If a namespace with that prefix is open for that node, the `Namespace` reference will be returned:

```
var somenode:XML = MXML.*::Slider[0]; //get the Slider node
var sNS:Namespace = somenode.namespace("s");
trace(sNS.prefix, sNS.uri); //s library://ns.adobe.com/flex/spark
```

Here, we get the `s` namespace out of the `Slider` node. This node is actually defined in the `mx` namespace, but it still has access to the `s` namespace as well as `fx`. Every node in the document has the same namespaces in scope because they are all declared on the document's root node. In other words, we can retrieve any namespace declared in the document on any node from the document.

Perhaps a better way to get declared namespaces out of XML is to retrieve them all at once with the `namespaceDeclarations()` method. This method takes no arguments and returns an `Array` of all XML namespaces open for the node. This is particularly useful because you don't need to know the prefix of a namespace to grab it:

```
for each (var ns:Namespace in MXML.namespaceDeclarations()) {
  trace(ns.prefix, ns.uri);
}
//fx http://ns.adobe.com/mxml/2009
//s library://ns.adobe.com/flex/spark
//mx library://ns.adobe.com/flex/halo
```

As you've seen, there are multiple ways to create, use, and retrieve namespaces. Which one you use will be determined by the kind of XML data you're using, how you use it, and your own personal coding style.

Additional Namespace Operations

E4X provides a few more namespace-related methods. Once you get the hang of creating namespaced XML, these should be straightforward additions to your E4X arsenal.

- `addNamespace()` and `removeNamespace()` — Allow you to add namespaces to your objects when dynamically constructing XML.
- `setNamespace()` — Sets the namespace of an XML object to the one you provide.
- `localName()` — Like the `name()` method, returns the node name for an element but in this case, the namespace prefix is omitted and only the local name is returned. For example, `<foo:bar/>.localName()` would return only "bar".
- `setLocalName()` — Likewise, sets the local name for an element with a namespace attached.

For further documentation on these methods, as always, consult the AS3LR.

Setting XML Options

We've already looked at the various XML parsing options made available through static properties of the XML class. These are:

- ignoreWhitespace
- ignoreComments
- ignoreProcessingInstructions
- prettyPrinting
- prettyIndent

In addition, these settings can be accessed as a bundle with XML's static settings(), which returns an associative-array Object with properties of the same names:

```
trace(XML.settings().ignoreProcessingInstructions); // Displays: true
XML.ignoreProcessingInstructions = false;
trace(XML.settings().ignoreProcessingInstructions); // Displays: false
```

You can also get a list of all the default values for these properties by using the defaultSettings() static method. There is also a setSettings() static method that allows you to change the settings object to a new Object. Although this allows you to change all values for the various settings, we recommend setting the individual properties to change these values. The one exception is if you choose to use the setSettings() method in conjunction with the defaultSettings() method to reset all the settings to their defaults.

In the following example, we use the defaultSettings() object to reset an individual flag, ignoreWhitespace, and then reset all the settings to their defaults using setSettings().

```
XML.ignoreWhitespace = XML.defaultSettings().ignoreWhitespace;
trace(XML.ignoreWhitespace); // Displays: true
XML.setSettings(XML.defaultSettings());
// All setting are returned to default
```

Note
Keep in mind that these settings are, for the most part, static properties and methods of the XML class and affect all XML objects. ∎

Summary

- E4X is the preferred, and standards-compliant, method of using XML in ActionScript. It encompasses both methods and special operators and statements recognized within ActionScript code.
- You can declare XML data directly in your code with XML literals, including using inline expressions.
- The XML class can represent an element, text node (including an attribute value), processing instruction, or comment. The XMLList class contains zero or more XML instances.
- Use the operators ., .., and @ to create concise XML queries. Values can be inserted as well as modified with E4X expressions. Nearly all the operators have method equivalents. These are summarized in Table 11-1.

- Filter XML data for items that match a particular set of criteria using a Boolean statement enclosed in the `()` E4X operator.

- By using `appendChild()`, `prependChild()`, `insertChildAfter()`, and so on, you can insert XML nodes into existing structures.

- Converting to and from strings can be achieved with the `XML()` constructor and the `toString()` and `toXMLString()` methods.

- Work with namespaces by using the `Namespace` class and the `namespace` keyword. Use the `::` operator to access an element in a specific namespace. Open namespaces with `use namespace`, and set the default namespace with `default xml namespace`.

- Gather meta-information about an XML node or document using the `nodeKind()` and `hasComplexContent()` methods.

- Global settings such as `ignoreComments` can be set using class properties or by accessing the `settings()` object of the XML class.

Regular Expressions

A lot of the work a program does is in getting and processing information from an outside source, no matter whether the source is a file, a web site, a service, a stored object, or user input. You have to extract or manipulate the information that the input contains, so ActionScript 3.0 gives you specialized tools for working with certain kinds of data. In Chapter 11, "XML and E4X," you saw that ActionScript 3.0 has a special syntax for dealing with XML data, so when you get XML from an external source, you can count yourself lucky and use E4X to interpret and manipulate it easily. A lot of the time you aren't so lucky, and the input you have to deal with uses a format ActionScript 3.0 doesn't support, or it's free-form. To help you deal with any kind of textual data, ActionScript 3.0 includes native support for regular expressions.

Introducing Regular Expressions

Regular expressions are fantastically cool. They are little programs inside your own program, written in their own concise language. You can store them and use them over and over. You can use them to search through text, extract just the parts you want from a bigger piece of text, or replace parts of text.

With regular expressions, you can handle any kind of text with a regular grammar, from HTML to LaTeX, Markdown to DTDs. You can use these expressions to create *lexers*, programs that break down a blob of text into discrete elements, like breaking an English sentence into words and punctuation. You can also use regular expressions to create parsers, programs that extract the meaning from text. As you've seen, ActionScript 3.0 contains built-in parsers for numbers, URL-encoded variables, CSS, and XML, enabling you to convert the string `"3.14"` to a `Number` type and the string `"<root><foo>bar</foo></root>"` to an `XML` type.

Most (application layer) protocols on the internet use text to communicate. For example, your mail client probably talks to the server using POP, SMTP, or IMAP, using simple commands and responses such as `"LIST"`, `"RETR"`, and `"OK"`. Even your web browser uses simple textual commands such as

`"GET /index.html"`. With sockets (introduced in Chapter 28, "Communicating with Remote Services"), you can use regular expressions to help you talk to a variety of services on the internet.

Quite a few languages have support for regular expressions, so you may have some experience with them. Regular expressions use the same syntax with little variation between languages that use them, so they are a skill you can apply and reapply.

Because regular expressions have been around for a long time without changing much, there is also an abundance of literature on them, if you wish to continue your studies beyond this chapter. An excellent resource is *Mastering Regular Expressions, Third Edition* by Jeffrey Friedl (O'Reilly, 2006).

Writing a Regular Expression

Regular expressions in ActionScript 3.0 are instances of the `RegExp` top-level class. This means you don't have to import anything to use `RegExp` objects. Just like `Numbers` and `XML`, regular expressions can also be written as literals. Sometimes you will want to use the constructor and sometimes the literal form. Let's get started and look at the general form of a regular expression:

```
var re1:RegExp = /sunken treasure/g;
var re2:RegExp = new RegExp("sunken treasure", "g");
```

The preceding two lines create identical regular expressions, first as a literal, and then using the `RegExp` constructor. Both `re1` and `re2` are of type `RegExp`. The literal form of regular expressions is also consistent between ActionScript and other languages: the pattern is written between forward slashes, followed by an optional combination of flags. In some languages the literal form executes the expression, whereas in ActionScript it only declares the expression. I'm starting out with a simple regular expression, one that searches for the text "sunken treasure" in every place that it might appear.

`RegExp` objects are built from the regular expression itself, in this case `sunken treasure`, and zero or more flags indicating how to apply the expression, in this case `g`, which applies the expression globally. You'll learn more about how to construct expressions and what the flags mean in the course of this chapter.

Applying Regular Expressions

Before you start building complex expressions, let's find out how you can apply regular expressions. Once you know how regular expressions can be used, you'll expand your repertoire of regular expression techniques to achieve specific goals. With an understanding of how they are applied, your knowledge of building them will have more meaning.

Earlier, I said that regular expressions can be used to lex and parse textual grammars. That's a high-level application. First you have to ask, what can you do with one regular expression?

String Methods and RegExp Methods

In these applications, you'll see that there are sometimes two ways to conduct the same kind of operation. The `String` class provides many functions that work with regular expressions, and the `RegExp` class provides functions that work with strings. Your decision on which to use might be based on functionality that only one of the functions provides, or it might be based on the structure of your code and what you want to achieve.

For example, if you were searching a thousand strings for a specific substring, you might create one `RegExp` object to find the substring and call `RegExp` methods on it, feeding it different `Strings` in a loop. If you were searching one string for a thousand different substrings, you might create one `String` object and call `String` functions on it, feeding it different `RegExp` objects.

As you see sets of functions that do similar things, think about these different cases, and it will become clear which one to use in which case. Rather than list these methods in an arbitrary order, I'll take some time with each kind of task you can achieve with an expression. In some cases, one method can be used for multiple purposes, so I'll show how to use some methods more than once (but in different ways).

Note

Because these methods exist in two different classes, and I'm mixing them up, I sometimes use the scope resolution operator (`::`) to clarify which class a method is associated with. For example,

```
Array::pop()
```

is the `pop()` method of an `Array` instance, whereas if it were written

```
Array.pop()
```

this would imply that `pop()` is a static method of the `Array` class. I've used this notation sparingly, but you should know what it means. ∎

Testing

You can use regular expressions to test if some text matches a certain pattern. Particular regular expressions are often called "patterns" because they represent certain patterns of text. In the first example, this pattern was a simple phrase ("sunken treasure"), but the pattern can be quite complex, to the point that you can't describe it easily in English and the regular expression that describes it is preferable.

Testing can also be achieved by paying attention to the result of a find or match operation. By way of analogy, if I asked you how many raisins were in your cereal, your answer would clearly indicate to me whether your cereal even contained raisins. Sneaky, right?

```
RegExp::test(str:String):Boolean
```

The `RegExp` method `test()`, when called on a regular expression, will run that expression on the passed string and return whether the string matches the pattern, as shown in Example 12-1.

`http://actionscriptbible.com/ch12/ex1`

Matching a Pattern

```
var phoneNumberPattern:RegExp = /\d\d\d-\d\d\d-\d\d\d\d/;
trace(phoneNumberPattern.test("347-555-5555")); //true
trace(phoneNumberPattern.test("Call 800-123-4567 now!")); //true
trace(phoneNumberPattern.test("Call now!")); //false
```

Example 12-1 creates an expression for numbers formatted as XXX-XXX-XXXX. It shows that regular expressions can match patterns of text, more than just looking for substrings. When you run this expression, it looks inside all the text for a match, so the second call to test() still finds the phone number, even though other characters are in the string.

Testing for the existence of a pattern finds out only if that pattern exists. You don't know from this example what the phone number is or where it appears in the text.

Locating

You can find a certain pattern inside a string and use its location for other string operations, such as splitting up the string or inserting new text — although you'll see that you can often perform these operations in the same step. You can locate the first instance of the pattern or everywhere the pattern appears.

```
String::search(pattern:*):int
```

Call search() on a String to find the first occurrence of the pattern you pass in, as shown in Example 12-2. You can pass it either a String or a regular expression, which is why the pattern argument is left untyped. This method searches from the beginning of the String to the end, starting at the beginning. If there are no matches, search() returns -1, making this an alternate way to test for existence of a pattern.

`http://actionscriptbible.com/ch12/ex2`

Collected Snippets: Locating a Pattern

```
var themTharHills:String = "hillshillshillsGOLDhills";
trace(themTharHills.search(/gold/i)); //15
```

This snippet sees if there is gold in them thar hills. Indeed, there is, starting at the fifteenth character: after three "hills," which have five characters each. If you were just searching for the lowercase string "gold," that would be too boring; after all, you learned how to search for substrings in Chapter 6,

"Text, Strings, and Characters." Instead, you passed it a regular expression that matches any pattern of the letters g, o, l, and d in either case, using the flag for case insensitivity, i. This matched the uppercase "GOLD." Not very well hidden, that gold. It would have also matched "gOLd," "GOLd," and the other 13 permutations of capitalizations.

```
RegExp::exec(str:String):Object
```

The exec() method of RegExp is a useful one, as Example 12-2 shows. You can use it to locate and identify patterns, and it plays nicely with loops. First let's use it to locate a pattern:

```
var searchForGold:RegExp = /gold/gi;
var themTharHills:String = "hillsgoldhillshillsGOLDhills";

var result:Object = searchForGold.exec(themTharHills);
//TEST if there's gold in the hills
if (result) trace("There's gold in them thar hills!");
//LOCATE the gold
trace(result.index); //5

//look a second time
result = searchForGold.exec(themTharHills);
//output the second location
trace(result.index); //19
```

You can see you're already getting more intricate. Here you have created a regular expression that finds the word "gold" in any combination of uppercase and lowercase in the string you pass it. It is also configured to match all instances of the pattern.

In the second paragraph, you execute the regular expression once. The expression goes through the string from beginning to end and stops when it finds the first match, the lowercase "gold" after the first "hill." The exec() method returns null if there are no (more) matches of the pattern; otherwise it returns an object with lots of useful data. The return type of exec(), and the variable result it is assigned to, is Object. First, you use this Object in a Boolean context to see if there are any matches of the pattern. Remember that by finding out *where* the gold is, you discover whether there is gold at all. Also recall that when coercing types to Booleans, objects are coerced to true when they contain any value, that is, when they are not null.

Next you use the index property of the object exec() returned. As you might guess, this property contains the location in the string of the current match. It tells you where the first instance of the "gold" pattern appears in the text. Note that I said "current" match. RegExp objects maintain one bit of state information; they remember the last position they matched so that you can repeatedly apply them to the same text.

In the third paragraph of the snippet, you run the expression again on the same string. It starts from the position of the first match and finds a second match: "GOLD." Running exec() with the same parameters gave you two different results, which means that exec() is a stateful method; it depends on the state of the RegExp you call it on.

Identifying

When you are looking for "gold," you know exactly what you're looking for and you're interested in where to find it. Many times you don't know precisely what you are looking for, only that it fits a certain pattern, and the contents of the matching text is of equal or greater importance than its position in the source text.

```
RegExp::exec(str:String):Object
```

You just saw how to use RegExp's exec() method to locate patterns in a string, so let's use it to identify them. When the exec() method finds a match, the Object it returns also includes the text that matched it. Let's use the phone number pattern from before to find the phone numbers in a big chunk of text, in Example 12-3.

EXAMPLE 12-3 `http://actionscriptbible.com/ch12/ex3`

Collected Snippets: Identifying a Match

```
var contactNumbers:String =
    "Call us at one of these numbers.\n" +
    "Los Angeles: 310-555-2910\n" +
    "New York: 212-555-2499\n" +
    "Boston: 617-555-7141";
var phoneNumber:RegExp = /\d{3}-\d{3}-\d{4}/g;

var result:Object;

while(result = phoneNumber.exec(contactNumbers)) {
    trace(result[0]); //prints out the phone numbers one by one
}
```

The snippet uses the same trick I just showed to loop through all the matches progressively. When there are no more matches, exec() returns null, so the expression that is the loop condition will be null, which is coerced to false, ending the loop. Inside the loop, you trace out the matched text itself. The example will trace out the three phone numbers from the source text in order.

The result object returned by exec() contains indexed properties as well as named properties, and the first index, result[0], contains the text that matched the pattern. This is how you use exec() to identify the match.

```
String::match(pattern:*):Array
```

The match() method of a String will return strings that match the pattern passed to it, as Example 12-3 shows. If the pattern is set to search globally, it returns all the matches; otherwise, it returns the first one. It, too, returns null when there are no matches.

The differences between this method and `RegExp::exec()` are that this method is called on the text and not the pattern, that it returns only the matched text, and that it runs all at once, rather than in a loop.

```
var contactNumbers:String =
    "Call us at one of these numbers.\n" +
    "Los Angeles: 310-555-2910\n" +
    "New York: 212-555-2499\n" +
    "Boston: 617-555-7141";
var matches:Array = contactNumbers.match(/\d{3}-\d{3}-\d{4}/g);
if (matches) {
    trace(matches.length); //3
    trace(matches[0]); //310-555-2910
}
```

Example 12-3 checks to see whether there are matches, and if there are, it traces out the number of matches and the contents of the first matching string. As you know, there are plenty of ways to manipulate array data, so it can be more convenient to get all the matches as an `Array` and then post-process it as necessary.

Extracting

Often, you need to find information from source text that appears in a certain context. You can add this context to a regular expression and indicate that certain parts of the pattern are to be captured for later use. Just as you can express the pattern you are looking for as extremely specific (the word "gold") or generic ("three numbers, a dash, three numbers, a dash, four numbers"), you can include any kind of context as well as any kind of pattern to capture out of the context, as shown in Example 12-4.

EXAMPLE 12-4 `http://actionscriptbible.com/ch12/ex4`

Collected Snippets: Extracting Matching Text

```
var contactNumbers:String =
  "Call us at one of these numbers.\n" +
  "Los Angeles: 310-555-2910\n" +
  "New York: 212-555-2499\n" +
  "Boston: 617-555-7141";
var nyPhone:RegExp = /New York: (\d{3}-\d{3}-\d{4})/;
var matches:Array = contactNumbers.match(nyPhone);
trace(matches[1]); //212-555-2499
```

This snippet creates an expression for text that follows the pattern of `New York: XXX-XXX-XXXX`. By including "New York:" in the expression, you know it's only going to match the New York phone number, and only if it is followed by a phone number in that format.

However, in the snippet you are concerned only with the actual number that you should call to get the New York office. You don't want "New York:" to come back, only the number itself; on the other hand, you need to include context in the expression so that it matches only the New York number. You achieve this by surrounding the number part of the expression in parentheses. This creates a capturing group. Any parts of the expression in parentheses like this will be specifically extracted for every match of the full expression. I'll cover capturing groups in detail later in the chapter.

Capturing groups are part of the expression, but you still need to run the regular expression to access these captures in a match.

```
String::match(pattern:*):Array
```

In the preceding snippet, you used `match()` to extract the captured group. However, the behavior of `match()` can be tricky here.

When the expression passed to `match()` has the global flag set, `match()` returns an array of all the substrings that match the pattern. It does not give you any of the captures. You'll read more on the global flag in the section "Regular Expression Flags."

When you use `match()` with an expression that does not have the global flag set, it returns an array containing the first matching substring, followed by all the capture groups in the expression for that match. This is the case in the preceding snippet. The expression does not have the global flag set, and `match()` returns an array where the first entry is the matching substring `"New York: 212-555-2499"`. The second entry (`matches[1]`) is the part of the expression that was captured, `"212-555-2499"`.

Because of this tricky difference, if you need to use capture groups, you might do well to stick to the by-now familiar function I revisit in the following section.

```
RegExp::exec(str:String):Object
```

Let's modify this program using `RegExp`'s `exec()` method so that you can get both the names and the numbers for every office in the list in one fell swoop. You'll add them to an associative array so you can look up numbers such as `contacts["New York"]`.

In Example 12-4, you're parsing arbitrary text with a specific format into a data structure that you can conveniently use in other code. That's pretty useful!

```
var contacts:Object = new Object();

var contactText:String =
    "Call us at one of these numbers.\n" +
    "Los Angeles: 310-555-2910\n" +
    "New York: 212-555-2499\n" +
    "Boston: 617-555-7141";
```

```
var officeAndPhone:RegExp = /^([\w\s]+): (\d{3}-\d{3}-\d{4})/gm;
var result:Object;
while (result = officeAndPhone.exec(contactText)) {
    contacts[result[1]] = result[2];
}
trace(contacts["New York"]); //212-555-2499
trace(contacts["Boston"]); //617-555-7141
```

The expression got a little more complex in this example, but don't worry about the specifics of the pattern. The expression matches the start of a line, a series of letters and spaces (which is captured), a colon and a space, and then a sequence of 3-3-4 digits (which is captured). It is set to match globally (meaning you couldn't use match() if you wanted to see the captured groups), and it is treated as a multiline string. I'll get into the details of how to build expressions like this shortly.

You use the RegExp exec() method again, and its multipurpose return object. The object contains the matched expression in the first indexed property, followed by all the captures for that match. So you use the first capture (the name of the office) as the key and the second capture (the phone number) as the value, adding them into the associative array contacts as you go. When the expression runs out of matches, the loop terminates, and you can successfully look up phone numbers in the contacts object by name.

Replacing

Once you have the ability to match patterns in text, replacing those patterns with desired text is a simple enough leap.

```
String::replace(pattern:*, repl:Object):String
```

You might recognize this function, as well, from Chapter 6. You can use this method with a string pattern and a string replacement object to perform a basic substitution. Why did I write this book anyhow? See Example 12-5.

EXAMPLE 12-5 http://actionscriptbible.com/ch12/ex5

Collected Snippets: Replacing Matching Text

```
var quote:String = "I did it for the money and the ladies.";
quote = quote.replace("money", "learning experience");
quote = quote.replace("ladies", "sleep deprivation");
trace(quote);
//I did it for the learning experience and the sleep deprivation.
```

Now that's closer to the truth! Remember that the `replace()` method returns the modified `String`, rather than modifying it in place, so you assign the result back to `quote` to immediately apply the replacements.

Now let's try replacing a pattern with a string.

```
var document:String = "My name is John Doe, SSN 123-45-6789," +
    "I live at 120 Birch St...";
var ssn:RegExp = /\d{3}-\d{2}-\d{4}/g;
document = document.replace(ssn, "<SSN REMOVED>");
trace(document);
//My name is John Doe, SSN <SSN REMOVED>, I live at 120 Birch St...
```

The example here searches for all patterns that look like a social security number (SSN) and sanitizes the document so that it contains no SSNs but indicates that they have been removed.

If you have capture groups inside your expression, when replacing text you can use the captured text as a variable inside the replacement string. For example, let's change all instances of "my name is X" to "X is my name."

```
var introduction:String =
"My name is Roger; but I'll have you know MY name is Henry!";
trace(introduction.replace(/my name is (\w+)/gi, "$1 is my name"));
//Roger is my name; but I'll have you know Henry is my name!
```

You have to search for the whole pattern "my name is" because you'll be modifying that phrase. However, of particular interest is the name itself, so you capture that. By using variables named $1, $2, and so on up to $99 in the replacement string, the first, second, and ninety-ninth captured groups are resubstituted into the text that replaces the match.

To make replacing text even sweeter, you can pass `replace()` a replacement *function* instead of a replacement string. The function is passed the matching text, all the captured groups for that match, and the full string in context, and you can make whatever kind of replacement you want. For instance, you could make sure that my friends' names always appear capitalized properly.

```
var friends:RegExp = /aashoo|betty|cesar|deepa|eve|frej|gina|hilary|isabell/gi;
var fixCaps:Function = function fixCaps(...args):String {
    var match:String = String(args[0]);
    return match.charAt(0).toUpperCase() + match.substr(1).toLowerCase();
}

var txt:String = "DEEPA is in SF cESaR is in Bogota and isaBELL is in Berlin.";
trace(txt.replace(friends, fixCaps));
//Deepa is in SF Cesar is in Bogota and Isabell is in Berlin.
```

You can do all kinds of things with custom substitutions, such as encoding particularly secret bits of information in an otherwise innocuous stream of text.

Splitting

You can use regular expressions to break up text. This is useful when you are trying to interpret text that uses a delimiter to denote breaks between data.

```
String::split(delimiter:*, limit:Number = 0x7fffffff):Array
```

This method was used with `String` delimiters in Chapter 6 to split text into its constituents. The `delimiter` parameter, however, may also be a regular expression. You can use this to allow for more flexibility in the delimiter. In this example, a haiku is split into phrases with slashes. However, you can't be entirely sure if the author will include spaces around the slashes, so you allow for whitespace on either side by splitting on a regular expression, as shown in Example 12-6.

Splitting Text

```
var haiku:String =
"Oh ActionScript three/ you make coding delightful / I think you are neat.";

var lineDelimiter:RegExp = /\s*\/\s*/;
var lines:Array = haiku.split(lineDelimiter);

for (var i:int = 0; i < lines.length; i++) {
    trace(i, lines[i]);
}
//0 Oh ActionScript three
//1 you make coding delightful
//2 I think you are neat.
```

The first and second delimiters in the example have different formats, but the regular expression catches them both. Additionally, by interpreting the whitespace (represented by \s, as you'll learn in the next section, "Constructing Expressions") as part of the delimiter, it is not included in the split-up substrings. As you can see, all three lines print out without extra whitespace in the beginning.

Constructing Expressions

By now, you've seen many examples of what regular expressions can be used for. Along the way you've also seen some pretty complex regular expressions. This section covers the building blocks of regular expressions and the flags they can be used with.
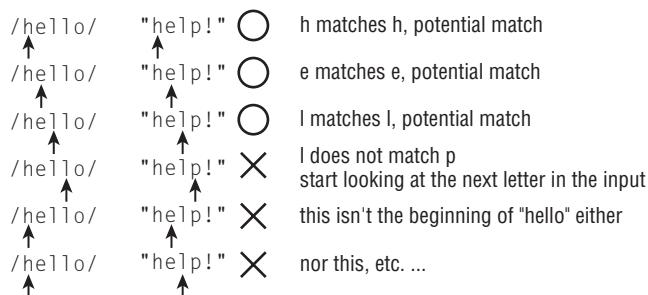
Each expression is actually a mini-program that can be run on a string input. In general, it steps through the string from start to end, trying to match the current part of the expression with the current part of the string, as you can see in Figure 12-1.

Stepping through a regular expression.


```
/hello/    "help!"  ○   h matches h, potential match
/hello/    "help!"  ○   e matches e, potential match
/hello/    "help!"  ○   l matches l, potential match
/hello/    "help!"  ✕   l does not match p
                        start looking at the next letter in the input
/hello/    "help!"  ✕   this isn't the beginning of "hello" either
/hello/    "help!"  ✕   nor this, etc. ...
```

Normal Characters

When you type letters in an expression, they match those letters in the same sequence, as shown in examples throughout this chapter. You can type any letter or number, and *some* punctuation, as shown in Example 12-7.

EXAMPLE 12-7 `http://actionscriptbible.com/ch12/ex7`

Collected Snippets: Constructing Regular Expressions

```
trace("are you thinking what I'm thinking?".match(/think/g));
//think,think
trace("rub a dub dub, three men in a tub".match(/ub/g));
//ub,ub,ub,ub
```

Dot Character

The period (`.`) has an important meaning in regular expressions. It matches *any* character, with one exception: the newline character. (A certain flag changes this behavior, covered in the section "Regular Expression Flags.")

Escaped Characters

There are two reasons to escape certain characters. First, in the language of regular expressions, many punctuation characters have special meanings. Second, escaped characters let you include characters that may be difficult to type or see in an expression.

I introduced character escaping in Chapter 6. To escape a character, simply preface it with a backslash (\). This goes for the backslash character as well. To type a backslash, you must prefix it with a backslash:

```
trace("c:\\windows\\"); //c:\windows\
```

You can't go wrong by escaping any character you type in a regular expression that is not an alphanumeric character (a–z, A–Z, and 0–9). It's not always necessary, but it's harmless:

```
var re:RegExp = /I have a \$5 bill \& his\/her ID card\./;
```

At a minimum, you must escape the following characters to ensure that they will be interpreted literally:

```
^ $ \ . * + ? ( ) [ ] { } |
```

The forward slash is added to this list, of course, if you are declaring a regular expression literally, because the forward slash marks the beginning and end of the expression.

There are also several escape characters that are written with a sequence but represent a different character than the character after the slash. These were covered in Chapter 6 and are summarized here in Table 12-1.

TABLE 12-1

Escape Sequences

Type This	Get This
\b	Backspace
\f	Form feed. This ejected the page currently in the printer back in the day.
\n	Newline, aka "LF." The line separator on UNIX machines and Mac OS X is \n
\r	Carriage return, aka "CR." The line separator in Windows is \r\n and on old Mac OS is \r.
\t	Tab.
\u*nnnn*	The Unicode character with character code *nnnn* in hexadecimal. For example, \u20ac is character U+20AC, the Euro sign (€).
\x*nn*	The ASCII character with the character code *nn* in hexadecimal. For example, \xa3 is ASCII character 0xA3, the pound sign (£).

Dealing with newlines on text that might come from different systems can be incredibly frustrating. Please keep the different possible line endings, as well as the encoding of the input text, in mind. See more on international regular expressions later in the chapter in the section "International Concerns."

Metacharacters and Metasequences Demystified

You may see the words "metacharacter" and "metasequence" when reading about regular expressions. These terms sound fairly impressive, but they have a simple meaning.

Both of these terms refer to any part of the pattern that has a special meaning in regular-expression-speak. A *metacharacter* is one single character that is not interpreted literally, such as *, whereas a *metasequence* is a sequence of characters that is not interpreted literally, such as \s or [a-z]. The asterisk does not mean "match an asterisk," and \n does not mean "match a backslash followed by the letter n." Escaped characters, therefore, are one kind of metasequence.

Character Classes

Regular expressions allow you to conveniently match a whole set of possible characters with a single metasequence. Several sequences look like escaped characters but actually represent a whole range of characters.

You already used a character class to match a phone number. Let's look at this expression in a little more detail:

```
var phoneNumberPattern:RegExp = /\d\d\d-\d\d\d-\d\d\d\d/;
```

Each instance of \d matches any single digit, meaning that except for the dashes, every character in the match must be a 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. You can easily remember this sequence because "d" is for "digit."

If you want to find any character that's *not* a digit, you can use the inverse of that character class, \D. The lowercase version matches digits; the uppercase matches nondigits.

There are three helpful shorthand character classes: \d for *digits*, \w for alphanumeric *word* characters, and \s for white*space*. Each of these has a capitalized version for its inverse: \D for nondigits, \W for nonword characters, and \S for nonwhitespace. The details of these are summarized in Table 12-2.

You can also create your own groups of character classes. Simply place the letters that you want to match inside square brackets:

```
trace("the cat sat on the mat".match(/[msc]at/g)); //cat,sat,mat
```

You don't need to separate the letters with anything, and their order doesn't matter. Keep in mind that even though the metasequence for the character class ([msc]) is longer than one character, it matches one character. It says "look for a character that is in this group." You can also specify ranges of characters with the dash character (-), and you can have multiple ranges in one character class. Furthermore, you can combine ranges with single characters:

```
trace("abcdefghijklmnopqrstuvwxyz".match(/[a-cmx-z]/g));
//a,b,c,m,x,y,z
```

Finally, you can invert a character class by using a caret (^) immediately after the open bracket ([). The whole sequence, then, matches a character that is *not* in the specified set:

```
trace("roger dodger".match(/[^oge\s]/g)); //r,r,d,d,r
```

In the preceding snippet, you find any letter that is not an o, g, e, or any kind of whitespace. Notice that you used a shorthand character class inside a character class. This, too, is possible.

When typing character sets, some characters must be escaped. Because the characters [, -, and] have special meaning in character classes, if you want to include these in a set, you must escape them. The caret may be included literally without escaping as long as it does not appear first.

TABLE 12-2

Character Classes

Character	Meaning
[...]	A character in the set (...).
[^...]	A character not in the set (...).
[x-y]	Any character that lies between x and y, inclusive.
\w	Word characters. Equivalent to [a-zA-Z0-9_].
\W	Nonword characters. Equivalent to [^a-zA-Z0-9_].
\s	Whitespace characters: tab, space, newline, carriage return. Equivalent to [\t\n\r] (note the space before the right bracket).
\S	Nonwhitespace characters. Equivalent to [^\t\n\r].
\d	Decimal digit characters. Equivalent to [0-9].
\D	Nondigit characters. Equivalent to [^0-9].
.	Any character.

Quantifiers

Regular expressions allow you to specify not just kinds of characters that match, but how many of them to match. Matching a specific number of characters is a useful ability, as is the ability to discard arbitrary amounts of noise between two items you are interested in matching.

```
trace(/\w+:\s*\$\d+/.test("soup:          $40")); //true
```

The preceding snippet makes good use of the plus quantifier (+). It is placed after the part of the pattern it is to quantify, and like all quantifiers, it changes the number of that subpattern that may appear in the text to qualify as a match. The plus symbol means "one or more," so the preceding pattern reads "one or more letters (\w+), a colon (:), some or no whitespace (\s*), a dollar sign (\$), and one or more digits (\d+)," which the test string indeed matches.

To match an optional character or sequence, use the question mark quantifier (?). The expression will then match text that has the subpattern zero or one time:

```
var betterPhoneNumber:RegExp = /\(?\d{3}\)?-?\d{3}-?\d{4}/;
trace(betterPhoneNumber.test("(703)555-1234")); //true
trace(betterPhoneNumber.test("310-555-1515")); //true
trace(betterPhoneNumber.test("7245559090")); //true
```

Here, you upgrade the phone number regular expression to make all the dashes optional and to allow parentheses around the area code. Remember that you must escape the parentheses in this expression to interpret them literally.

To match any number of a character or sequence, including none at all, you can use the star (*) quantifier. This modifies the preceding part of the pattern to match zero or more times:

```
trace("a thousand thousandss!".match(/thousands*/g)); //thousand,thousandss
```

In this snippet, the "s" character can either not appear or appear any number of times. No matter how many s's are put at the end of "thousands," it will continue to match.

You can also specify an exact quantity for the subpattern to appear in. In the phone number regular expressions in Examples 12-3 and 12-4, you matched the exact number of digits. You can write expressions to match an exact number, a minimum number, or a specific range of acceptable numbers. These options are summarized with the quantifier characters in Table 12-3.

TABLE 12-3

Quantifiers

Quantifier	Meaning
*	Zero or more; any number
?	Zero or one; optional
+	One or more; required
$\{n\}$	Exactly n times
$\{n,\}$	At least n times; n or more times
$\{n,m\}$	Between n and m times, inclusive

Without quantifiers, an expression expects to match exactly one instance of each character.

Anchors and Boundaries

A few regular expression symbols help you anchor parts of the expression to interesting parts of the input text. These are different from the kinds of metacharacters and metasequences you've seen so far in that they match a *position* rather than a character.

You can specify where the beginning and end of a string or line is supposed to appear within a pattern by using the ^ and $ anchors, respectively. When the multiline flag m, covered in the next section, "Regular Expression Flags," is set, these anchors match the beginning and end of a line; without the flag, they match the beginning and end of the entire string.

Let's revisit the contact parsing expression from the previous section:

```
var contactText:String =
    "Call us at one of these numbers.\n" +
    "Los Angeles: 310-555-2910\n" +
    "New York: 212-555-2499\n" +
    "Boston: 617-555-7141";
var officeAndPhone:RegExp = /^([\w\s]+): (\d{3}-\d{3}-\d{4})/gm;
```

By setting the multiline flag on this expression and anchoring to the beginning of the line, you ensure that the "name: phone number" pairs are found on every new line. In this case, if you had two sets of name/phone pairs on one line, only the first one would match because the pattern matches only when it appears at the beginning of a line. You could disqualify that kind of line entirely by insisting that the line end right after the phone number:

```
var officeAndPhone:RegExp = /^([\w\s]+): (\d{3}-\d{3}-\d{4})$/;
trace(officeAndPhone.test(">>>New York: 212-555-2499")); //false
trace(officeAndPhone.test("New York: 212-555-2499 (fax)")); //false
```

These two test cases fail to match the pattern because they both contain garbage before and after the pattern, where the pattern specifies that it fills up the entire string. Anchoring your pattern to both the beginning and end of a string/line is common.

Because these symbols match positions rather than characters, they are *zero-width*: when the regular expression is moving through the pattern and one of these anchors matches, it doesn't move to the next character in the string. When the "h" in /hello/ matched the "h" in "help!" in Figure 12-1, you moved on to the next character in the string and the pattern. When you match the beginning of a string with ^, for example, you don't move to the next character in the string.

Additionally, you can use a metasequence that represents a word boundary. The anchor \b matches a position between word characters and nonword characters (as defined by \w and \W). This matches both the beginning and the end of words. The benefit to this character is that it is zero-width.

```
var usingBoundary:RegExp = /\bzarflax\b/i;
var noBoundary:RegExp = /zarflax/i;

var garbage:String = "asndzarflaxhtewio";
trace(usingBoundary.test(garbage)); //false
trace(noBoundary.test(garbage)); //true

var dialogue:String = "Nice try, Zarflax!";
trace(usingBoundary.test(dialogue)); //true
trace(noBoundary.test(dialogue)); //true
```

```
var fakingBoundary:RegExp = /\Wzarflax\W/i;
trace(garbage.match(fakingBoundary)); //null
trace(dialogue.match(fakingBoundary)); // Zarflax!
```

In the second paragraph of this snippet, you see that not using word boundaries matches the word "zarflax" even when it is contained within garbage, which in this case is a false positive. Adding word boundaries to the pattern causes the garbage string not to match. In the third paragraph, you test against the intended usage. Both with and without boundaries, the word is found.

In the final paragraph of the snippet, you can see why using a zero-width word anchor is different from including the surrounding nonword characters in the pattern. Doing so causes the garbage string not to match because there is no instance of "zarflax" that has nonword characters before and after it in the garbage string. So far so good — you've eliminated the false positive. But when you attempt to match the pattern in the dialogue string, you end up picking up the nonword characters you required "zarflax" to be surrounded in as part of the pattern. The benefit of \b here is that it matches a word boundary without consuming it into the pattern.

Table 12-4 summarizes the anchors available to regular expressions in ActionScript 3.0.

TABLE 12-4

Anchors and Boundaries

Anchor	Meaning
^	Beginning of string, or beginning of line when multiline flag is set
$	End of string, or end of line when multiline flag is set
\b	Word boundary; after nonword character and before word character
\B	Not word boundary; between two word characters or between two nonword characters

Alternation

Using a pipe (|) character in a regular expression allows you to match multiple alternatives. You saw this at play earlier in the expression:

```
var friends:RegExp = /leigh|mariko|neal|oskar|paula/gi;
```

You can use one pipe to allow two options, or multiple pipes to allow many options. This example matches the name of any of these friends. Alternation can be even more useful when you can specify parts in which alternates are allowed instead of alternating the entire expression. You do this with groups.

Groups

Grouping can be used for several purposes. You've already seen that it can capture information out of a specific context, using the pattern at large to match the entire format and the group to make the text you're really interested in available after the match.

Use parentheses around part of the pattern to make a group out of it. This kind of group is a *capturing group* because it is captured for later use.

Caution

If you want to include parentheses in your expression, to match actual parentheses in the input text, you must escape the parentheses. Use \(to match an open parenthesis and \) to match a close parenthesis. ■

You can also use groups as the container for alternates or repetition:

```
var rhymes:RegExp = /\b(tr|r|sp|b)a(c|s)e\b/gi;
var str:String = "trace() the race to the orbiting base in outerspace";
trace(str.match(rhymes)); //trace,race,base
```

This example goes further than a character class could. The (c|s) alternate could be replaced by [cs], but the beginning of a rhyming word can be one or two letters when you use alternates like this. In fact, an alternate can be any pattern, not just these simple letter sequences.

The two groups in this expression act as a scope for the alternates. If any of the alternates in the first group are fulfilled, the string and regular expression both advance to the next character. So you can look at the group as a subpattern, a single special kind of match that is self-contained.

Using groups for scoping purposes also allows you to apply quantifiers to subpatterns, as in the following example:

```
trace("the cat goes mew meow mewmew meow!".match(/(meo?w\s*)+/));
//mew meow mewmew meow,meow
```

This pattern tries to match a complete line of cat talk, crafting a subpattern to match a single cat word and then requiring one or more instances of that whole subpattern. The second argument in the returned array "meow" is included because, as you might recall, when calling match() without the global flag, it also returns the captured groups. In this case there is only one capturing group, but it is repeated, so when the matching is finished, that capture includes the last instance of the subpattern to be captured. The first argument is still the matching text in its entirety. You can see that the expression has successfully extracted all cat language from the string.

Regular Expression Flags

Flags modify the interpretation of a regular expression. You've already used flags extensively in the examples in this chapter. This section covers these flags and their effects in detail. The default is for all these flags to be off — that is, an expression with no flags set has all the flags off; you turn them on by adding them to the expression.

To recap, flags can be specified when the expression is created, either after the final slash in a literal `RegExp` or as the second parameter of the `RegExp()` constructor. The following snippet activates all five available flags, first using a literal and then the constructor:

```
/foo/gimsx;
new RegExp("foo", "gimsx");
```

These two expressions are equivalent. The flags in a regular expression may appear in any order and combination.

Global

The global flag `g` allows the expression to be used repeatedly on the source text until there are no more matches. When it is not set, the expression returns the first match.

This flag applies to `String::match()`, `String::replace()`, `RegExp::test()`, and `RegExp::exec()`. In `String::match()`, the global flag determines whether the method returns an array of the first match and captured groups (when it is not set) or an array of all matches (when it is set). In `String::replace()`, the flag determines whether the first match is replaced (when it is not set) or all matches are replaced (when it is set). In `RegExp::test()` and `RegExp::exec()`, if the global flag is set, the expression continues to match from the *n*th character in the source text, where *n* is the expression's `lastIndex` property. This property is set after every match so that you can use a loop to progressively step through all matches with `RegExp::test()` or `RegExp::exec()` when the global flag is set. If the global flag is not set, these methods match the first occurrence by starting matching at the first character.

The global flag does not need to be set to run `String::split()`. This method uses the `limit` passed in the second parameter to determine how many times the delimiter pattern may match. By default (when no `limit` parameter is passed), the string is split as many times as necessary, up to an implausibly large number.

Ignore Case

The ignore case flag, `i`, matches characters regardless of their case. This means that `/a/i` matches A or `a`. This flag affects all the methods I have covered.

You saw this flag used in the example where you fixed words with incorrect capitals:

```
var friends:RegExp = /aashoo|betty|cesar|deepa|eve|frej|gina|hilary|isabell/gi;
```

This way, the expression matched "cESaR" as well as "DEEPA."

Caution

Turning on the ignore case flag can make your expressions much more tolerant. This can be dangerous or help-ful depending on your intent. When parsing a natural language like English text, the meaning of the words is usually more important than the capitalization. When dealing with URLs or file paths, servers and operating systems are frequently case sensitive, and it's important to preserve case. ■

Multiline

The multiline flag, m, changes the behavior of the ^ and $ anchors in the expression. When the mul-tiline flag is off, these metacharacters anchor to the first and last character of the entire text. When the flag is set, they match at the beginning and end of every line. This flag affects the behavior of the expression regardless of the method using it.

This flag can be useful when parsing strings that contain multiple lines (no surprise there). The same effects can generally be reached by splitting the input text into an array of lines and then processing each line independently, as shown in Example 12-8.

EXAMPLE 12-8 `http://actionscriptbible.com/ch12/ex8`

Collected Snippets: Regular Expression Flags

```
var contacts:Object = new Object();
var contactText:String =
    "Call us at one of these numbers.\n" +
    "Los Angeles: 310-555-2910\n" +
    "New York: 212-555-2499\n" +
    "Boston: 617-555-7141";
var officeAndPhone:RegExp = /^([\w\s]+): (\d{3}-\d{3}-\d{4})/;

var lines:Array = contactText.split(/\n/);
var line:String;
for each (line in lines) {
    var result:Array = line.match(officeAndPhone);
    if (result) {
        contacts[result[1]] = result[2];
    }
}
```

You can simplify this code by using the multiline flag:

```
var contacts:Object = new Object();
var contactText:String =
    "Call us at one of these numbers.\n" +
    "Los Angeles: 310-555-2910\n" +
    "New York: 212-555-2499\n" +
    "Boston: 617-555-7141";
var officeAndPhone:RegExp = /^([\w\s]+): (\d{3}-\d{3}-\d{4})/gm;

var result:Object;
while (result = officeAndPhone.exec(contactText)) {
    contacts[result[1]] = result[2];
}
```

By using the multiline and global flags, the same expression can match multiple times, even though the source text spans across several lines.

Dotall

The dotall flag, s, changes the behavior of the dot metacharacter. When the flag is not set, the dot (.) matches any character but a newline character. When the flag is set, the dot (.) matches every possible character including newlines.

You can think of this mode as *single-line mode* because that evokes s much better than dotall and because it can be used to treat a string with multiple lines as one long string that happens to have \n characters in it. With the s flag on and the m flag off, the regular expression no longer maintains the illusion that the input string is partitioned into lines. You can see the dotall flag's effect in the following snippet:

```
var lines:String = "hello,\n cruel world!";
trace(lines.match(/hello.*world/)); //null
trace(lines.match(/hello.*world/s)); //hello,
                                     // cruel world
```

The first match() in this snippet operates on the assumption that you don't want your pattern to match across line breaks unless you explicitly include a line break character. The second match() allows anything to be in the .* subpattern, even newlines.

Caution

Be careful with .*. This pattern is greedy and can eat more of the input text than you desire. With s on, it can eat its way across lines and to the end. You can be more cautious with a lazy pattern. The next section, "Constructing Advanced Expressions," explains lazy and greedy patterns. ■

Extended

The extended flag, x, is included in an attempt to mitigate the general illegibility of regular expressions. Making dense, amazingly complex regular expressions that do the same work as dozens of

lines of code is, I can vouch, rewarding, but when it comes time to explain your code to someone else or even to remember what you wrote six months ago, you might see the other edge of this sword.

The extended flag lets you mitigate regex clutter by adding whitespace anywhere in the expression, which will not be literally interpreted unless it is escaped. When the flag is off, a space in a regular expression means "match a space in the source text in this position": The whitespace is taken literally.

Unfortunately, adding whitespace can do only so much for the understandability of your expressions. It's a good idea to briefly say what an expression does in a comment adjacent to it when the expression is complicated:

```
//find these complete words that rhyme with base: space, trace, race.
var rhymes:RegExp = /\b  (tr|r|sp|b) a (c|s) e  \b/gix;
```

Retrofitting the rhyme example with whitespace makes it a tad more legible. A comment helps more.

Table 12-5 recaps the regular expression flags that ActionScript 3.0 supports.

Regular Expression Flags

Flag Character	RegExp Property	Behavior When Flag Is Set
g	global	Don't stop after first match.
i	ignoreCase	All alphabetical comparisons are case insensitive.
m	multiline	^ and $ anchors match beginning and end of lines.
s	dotall	. matches newline.
x	extended	Whitespace in expression is ignored unless escaped.

Constructing Advanced Expressions

In this chapter, I introduced a lot of information in a short space. The regular expression techniques discussed thus far should serve for a majority of your text manipulation challenges. I did, however, leave out a few points about regular expressions, which build on the foundations I laid earlier.

If you are learning regular expressions for the first time, it might be helpful to return to this section after you've implemented a few regular expressions of your own and become comfortable with the way they work.

Greedy and Lazy Matching

The + and * quantifiers can be difficult to tame. You might have run into this before if you attempted to parse HTML, as the expression in Example 12-9 is intended to. (If the HTML is valid XHTML, you should use E4X for this and spare yourself the headache.)

EXAMPLE 12-9 `http://actionscriptbible.com/ch12/ex9`

Collected Snippets: Advanced Regular Expressions

```
//match <, (the tag name), any other stuff till >, >, (the inner text),
//then </, (a tag name), and >.
var getTag:RegExp = /< (\w+) \s* [^>]* >  (.*)  <\/ (\w+) >/x;
var parts:Array;

var htmlFragment1:String = '<li>item 1</li>';
parts = htmlFragment1.match(getTag);
trace("nodeName:", parts[1]); //nodeName: li
trace("innerHtml:", parts[2]); //innerHtml: item 1

var htmlFragment2:String = '<li>item 1</li><li>item 2</li>';
parts = htmlFragment2.match(getTag);
trace("nodeName:", parts[1]); //nodeName: li
trace("innerHtml:", parts[2]); //innerHtml: item1</li><li>item 2
```

The `getTag` expression works fine for a single tag. You want to be able to apply the expression over and over to pull out several tags in a row, but there's a problem. The `.*` in the middle of the expression keeps eating up everything until the last close tag. When the regular expression program sees the first closed tag that matches the last part of the expression, the `.*` remains in control. Its meaning, "any characters," certainly applies to `</li>` as well, so the `.*` in the pattern matches everything possible without preventing the rest of the expression from matching. The subexpression lets go before the last `</li>` so that the full expression can still match.

This behavior is called *greedy matching* because the subexpression, which has one of these quantifiers, greedily matches as far as it can into the string. By default, all regular expression quantifiers are greedy. If you could make the `.*` in the middle of the expression less greedy, the end of the regular expression might be able to match the first `</li>` instead of the last one, so that the whole expression matches only one `li` tag, the intended behavior.

Thankfully, there are more polite versions of these quantifiers. By adding a question mark to the quantifiers, you can make them match *lazily* instead of greedily. Lazily matching subexpressions stops

matching at the first opportunity to match the rest of the expression rather than the last. With the addition of this character, you can fix the regular expression:

```
//match <, (the tag name), any other stuff till >, >, (the inner text),
//then </, (a tag name), and >.
var getTag:RegExp = /< (\w+) \s* [^>]* >  (.*?)  <\/ (\w+) >/gx;
var htmlFragment:String = '<li>item 1</li><li>item 2</li>';
var parts:Array;
while(parts = getTag.exec(htmlFragment))
{
    trace("nodeName:", parts[1]);
    trace("innerHtml:", parts[2]);
}
//nodeName: li
//innerHtml: item 1
//nodeName: li
//innerHtml: item 2
```

Notice that the only difference in the regular expression (besides applying it globally) is the change from `.*` to `.*?`, which does the trick to match only until the first end tag.

You can use these lazy quantifiers any time you want to match a general pattern between two specific patterns, without accidentally including the ending pattern in the middle. Useful lazy quantifiers include `*?` and `+?`, although you can make any quantifier lazy.

Backreferences

Throughout this chapter, you have seen how to use capturing groups to store particular subexpressions and use them in code. Regular expressions also allow you to use the captured groups inside the same expression they are captured with.

You use a *backreference*, as it is called, by referring to the ordinal number of the matching group, like using $1 through $99 in `String::replace()`, except backreferences use backslashes instead of dollar signs: \1 through \99. The text that matches the captured group is used as the subexpression where the backreference appears instead of the literal characters \1. This snippet demonstrates:

```
var re:RegExp = /\b (\w{3}) \b  .*?  \b \1 \b /ix;
var testString:String = "far, farther fetched free fun fetched fun free";
trace(testString.match(re)); //fun fetched fun,fun
```

The preceding regular expression finds the first three-letter word that appears at least one more time anywhere in the text. It uses a group to capture a three-letter word, `.*?`, to keep matching through the rest of the string, and finally a backreference to assert that the word is repeated later in the string.

Because it uses the word boundary anchor \b, it also ensures that the second time the captured three letters appear, they are in a single word. This disqualifies the word "far," which appears as part of the word "farther." If the expression matches at all, the first repeated three-letter word is stored in the first captured group.

Lookahead and Noncapturing Groups

These different kinds of groups can be used for interesting purposes.

Noncapturing Groups

When you want to use a group for alternation or a quantifier, but the actual contents of the group are not as important to your expression, you can reduce the amount of unnecessarily captured groups by making a group that does not capture: a *noncapturing group*. This is achieved with the following syntax:

```
(?:...)
```

where your grouped subexpression takes the place of the ellipsis. For example:

```
var re:RegExp = /(?:i|you|he|she|it|we|they) likes? to (\w+)/i;
var match:Object = re.exec("We like to party.");
trace(match[1]); //party
```

This expression finds what it is that someone likes to do. It uses a group to allow for several subjects. However, the first group is noncapturing, so it does not interfere with the verb, which is captured in the first position.

Note
When using a noncapturing group, the contents of the group are still available to backreferences. ∎

Positive Lookahead Groups

Beyond noncapturing groups, there are also zero-width groups: groups that try to match the source string without moving ahead in the pattern. ActionScript 3.0 regular expressions have a *positive lookahead group*, or a *zero-width positive lookahead assertion*. When you use the syntax

```
(?=...)
```

you ensure that the pattern represented by the ellipsis would match, if you tried to. However, it doesn't perform the match, so that after the lookahead group, you're still talking about the same location in the source text as before it. This is unlike most metasequences you've seen so far, which match and then continue through the source text. Recall that anchors such as ^ and \b are also zero-width.

```
var re:RegExp = /\b (?:\w{3}) \b  (?= .*?  \b \1 \b) \s* (\w+) /x;
var testString:String = "far, farther fetched free fun fetched fun free";
trace(testString.match(re)); //fun fetched,fetched
```

Here you modified a prior snippet to find the word *after* the first three-letter word that repeats. The large lookahead in the middle looks ahead to see if there is, after a sequence of characters, another instance of the three-letter word matched. If this lookahead fails (if there is no duplication of the three-letter word just found), the potential match immediately fails and the next bit of source text is considered. When the lookahead succeeds, as it does with "fun," it must have examined the source text through the second instance of the word "fun." But when the lookahead group matches positively, the regular expression restores the current position in the source text to the position that was already being considered when the expression encountered the lookahead group: after the word boundary after the first "fun." Then the remainder of the expression is considered. After a lookahead group matches, the pointer in the expression is at the end of the lookahead group in the expression, but the pointer in the source text hasn't moved. It's zero-width. Thus, when the match continues at \s*, the source continues after the first "fun" and achieves the goal of getting the word after the first repeated three- letter word.

Incidentally, the snippet also uses a noncapturing group to find the three-letter word because the goal of the search is no longer the word, but the word immediately following. This way, the only captured group is "fetched," the solution to the puzzle. You can also see that the entire match "fun fetched" does not include text up to the second instance of "fun," even though that source text was examined by the lookahead.

Negative Lookahead Groups

In addition, you have negative lookahead groups which, like their brethren, do not advance the portion of the string being considered and do not capture. The difference, of course, is that they assert that the pattern inside them would *not* be matched at the current position. To write a negative lookahead group, use the syntax

```
(?!...)
```

with your subexpression in place of the ellipsis.

Named Groups

Another way to mitigate the jumble that can arise from complex applications of regular expressions is to assign names to capturing groups, rather than referring to them by their ordinal. Names are more descriptive than numbers, and as you can name variables descriptively in ActionScript 3.0, so can you assign names to capturing groups to clarify code dealing with regular expressions.

To create a named group, use the format

```
(?P<name>...)
```

where *name* is the name you to refer to the group if it is captured, and the expression to be captured replaces the ellipsis.

Named groups are applicable only to methods that would otherwise return the matched groups in an array: `RegExp::exec()` and `String::match()`. The names cannot be used in metasequences, either as backreferences or in `String::replace()`. When named groups are used, the captured groups are returned from these methods both in their ordinal position and by their named property.

Let's say that you want to create a method to convert *From:* e-mail headers into a more readable format. Sometimes these headers come with both the sender's real name and e-mail address, in the format "Roger Braunstein <roger@actionscriptbible.com>". If this is the case, you want to present the field as the sender's name, hyperlinked to send e-mail to the correct address when clicked (see Example 12-10).

EXAMPLE 12-10 `http://actionscriptbible.com/ch12/ex10`

Using Named Groups

```
package {
  import com.actionscriptbible.Example;
  public class ch12ex10 extends Example {
    public function ch12ex10() {
      var testLink:String = '"Roger Braunstein" <roger@actionscriptbible.com>';
      trace(headerToLink((testLink)));
      //<a href="mailto:roger@actionscriptbible.com">Roger Braunstein</a>
    }
    protected function headerToLink(fromField:String):String {
      //a from address with a name takes the format:
      //(some stuff, perhaps in quotes, which is the name),
      //perhaps some whitespace, <,
      //(some stuff with an @, which is the address), >
      var findHeader:RegExp = /"? (?P<name>.+?) "? \s* < (?P<email>.+@.+) >/x;
      var match:Object = fromField.match(findHeader);
      if (match) {
        return '<a href="mailto:' + match.email + '">' + match.name + '</a>';
      }
      //if this is not the format of the header, don't mess with it
      return fromField;
    }
  }
}
```

The class uses named capturing groups to refer to `match.name` instead of `match[1]`, making the code more self-explanatory. Table 12-6 summarizes the kinds of groups you've learned about in this section.

TABLE 12-6

Regular Expression Groups

Metasequence	Meaning
`(foo)`	Match ''foo'' and capture it.
`(?:foo)`	Match ''foo.''
`(?=foo)`	Continue only if ''foo'' would match here.
`(?!foo)`	Continue only if ''foo'' would fail to match here.
`(?P<bar>foo)`	Find ''foo'' and capture it with the name *bar*.

International Concerns

When you use regular expressions to deal with accented and non-English text, keep these tips in mind:

- Regular expressions, like strings, may contain UTF-8 encoded characters in the code. They match corresponding characters in the input text.

- Although there is a metasequence for Unicode characters, you may also include them literally in the expression.

- The shorthand character classes include only the English alphabet. For example, \w includes only *a* through *z*. No accented characters are included.

- When using the ignore case flag (*i*), only the characters *a* through *z* are case insensitive. For example, *é* does not match *É*, even with the ignore case flag set.

Using the RegExp Class

In the examples in this chapter, you have seen regular expressions written as literals. However, using regular expressions as `RegExp` objects enables you to apply some useful techniques and gain important insight.

Building Dynamic Expressions with String Operations

Keep in mind that if you use the `RegExp` constructor to create expressions, you can pass in an arbitrary `String` for the source of the pattern. This means that you can, using `String` manipulation, programmatically craft a pattern to fit the situation at hand, as shown in Example 12-11.

Using the RegExp Class

```
package {
  import com.actionscriptbible.Example;
  public class ch12ex11 extends Example {
    public function ch12ex11() {
      trace(removeLetters("catch", "cathy catnaps in cathay")); //y nps in y
      trace(removeLetters("bol", "Bob Loblaw Law Blog")); // aw aw g
    }
    public function removeLetters(inWord:String, fromWord:String):String {
      return fromWord.replace(new RegExp("[" + inWord + "]", "gi"), "");
    }
  }
}
```

The preceding example removes all letters found in a word from another bit of text. It does so by turning your blacklisted-letters string into a character class, dynamically generating the regular expression to remove all offensive letters.

RegExp Public Properties

Whether a RegExp object is created with the RegExp constructor or a literal, you can access the public properties that RegExp defines to recall information about the expression.

Keep in mind that all these properties but one are read-only. They are defined during construction and are not modified subsequently:

- source:String — The expression's source text.

- dotall:Boolean — Whether the dotall (s) flag is set.

- extended:Boolean — Whether the extended (x) flag is set.

- global:Boolean — Whether the global (g) flag is set.

- ignoreCase:Boolean — Whether the ignore case (i) flag is set.

- multiline:Boolean — Whether the multiline (m) flag is set.

- lastIndex:Number — The index of the character in the input text where the pattern should next begin matching. This property is read-write. This state variable affects only exec() and test() and is set by these methods when (progressively) matching a pattern with the global flag set.

Summary

- Regular expressions can be used to parse regular grammars and loose human input.
- Regular expressions are used to test, locate, identify, extract, replace, and split text.
- Regular expressions are a compact language defined by normal text, metacharacters, and metasequences.
- Flags change the behavior of regular expressions.
- Often you have to choose between a `String` method and a `RegExp` method to achieve the same goal.
- You can match characters, alternates, classes of characters, and groups, in whatever quantity you specify.
- Quantifiers are greedy unless you make them lazy.
- Anchors can match a position rather than text.
- Groups can be made to not capture, to be zero-width, and to have names.
- Case insensitivity and word characters work only as advertised with English letters from *a* to *z*.

Binary Data and ByteArrays

I n ActionScript and other programming languages, you use a variety of data types to intelligently organize data. Breaking out of the walled garden of your own program and accessing data from the outside world can force you to change representations of the data, since not every system uses the same data types or encodes them in the same way. These conversions can be automatic and effortless, or they can require painful acrobatics and careful handling of edge cases. The ultimate common denominator in representations of data, of course, is binary: the long digital sequences of 0s and 1s that are sent about your computer hardware, and across wires and radio signals that connect your computer to the internet.

By supporting high-level data types like `Vector` and `Dictionary`, Flash Player gives you the opportunity to create algorithms that tackle problems with the appropriate tools or to build those tools when they aren't included (like an AVL tree, a doubly linked list, or a directed graph, for example). But by supporting raw binary data, Flash Player opens up the entire universe of data. Without byte-level access, you can interpret data only if it comes in a format that Flash Player understands. With byte-level access, you can interface with any computer or interpret any digitized data if you can write an interpreter for it in ActionScript.

It's the inclusion of binary types such as `ByteArray` that allows Flash Player to capture video data and track a fiducial marker from your camera to display a 3D object in augmented reality. It's binary math that permits Flash Player to synthesize dynamic audio. And it's binary types that allow Flash Player to save out a PNG or generate a PDF. Binary access grants Flash Player the kind of universal reach that is typically reserved for desktop applications.

FEATURED CLASSES

`flash.utils.ByteArray`

`flash.utils.IDataInput`

`flash.utils.IDataOutput`

Binary Concepts

You probably have some idea of what binary is and how to work in it. I'll present a brief review here. If you have a good grip on binary and just want to see how to deal with it in ActionScript 3.0, skip ahead to the section "Binary Types in ActionScript."

You probably already know that binary means base two. There are two digits in base two: 0 and 1. Each place for a digit in binary is a power of 2 greater than the place to its right. So 1000 in base 10 is 10^3, and 1000 in base 2 is 2^3, or 8. To store whole positive numbers — uints — just convert the number from decimal to binary. For example, 19 in decimal is 10011 (16 + 2 + 1) in binary, and that's how the computer stores it. Chapter 7, "Numbers, Math, and Dates," investigated the binary representations of numbers.

We represent not just numbers but all data on a computer in binary. For example, we represent text in binary as a series of characters. There are various encodings of characters, with the simplest being ASCII, in which every letter in the English alphabet is assigned a number between 1 and 255. The number 0 is reserved to indicate the end of a string. For example, 72 is H and 73 is I, so you say "HI" with the series of numbers 72, 73, and 0, or 0x484900. This is Western-centric, so ActionScript 3.0 uses UTF-8 for characters. UTF-8 can use between 1 and 4 bytes for a single character, instead of always using 1. And it can encode practically every written language ever used on Earth. Also, 1-byte sequences in UTF-8 are identical to ASCII, giving it lots of compatibility. Chapter 6, "Text, Strings, and Characters," illustrates how UTF-8 encoding is used in Strings. Besides these two, however, there are many more character encodings in use.

A single binary number is called a *bit*. Instead of dealing with bits, however, you usually deal in bytes. A *byte* is a set of 8 bits, or a number between 00000000 and 11111111 in binary. There's not much you can do with a single bit, except maybe store a Boolean — because it can only be true or false, which you could assign the values 1 and 0. But these tiny sizes of information are too small to bother with for the computer's memory management, so you'll never see an object smaller than a byte. The truth is that a Boolean variable takes up 4 bytes (that's 32 bits) in Flash Player. By the way, a byte can also be called an *octet*, which is a pretty logical name if you think about it.

Of course, you're familiar with other sizes of data.

- byte — 8 bits
- kilobyte (KB) — 1024 bytes
- megabyte (MB) — 1024 kilobytes, or 2^{20} bytes
- gigabyte (GB) — 1024 megabytes, or 2^{30} bytes
- kilobit (Kbit) — 1024 bits, or 128 bytes
- megabit (Mbit) — 1024 kilobits, or 128 kilobytes

And so on. However, there is ongoing confusion about whether to use binary (2^{10}, or 1024) or decimal (10^{10}, or 1000) powers for the prefixes kilo-, mega-, giga-, and so on. Different measurements are used in different contexts, which makes the whole thing a complicated mess. For example, a hard drive manufacturer would benefit by calling a gigabyte 1000 megabytes, because then she could report a higher capacity for the same hardware than if she had called a gigabyte 1024 megabytes. Confusingly, data rates such as transfer speed are often measured in kilobits or megabits, while you're used to measuring things in kilobytes or megabytes. It's funny, but although digital data has been around for a long time, people are still figuring out how to consistently communicate sizes.

Because it's painfully long to write 1s and 0s — the ASCII character R is 01010010 in binary, and that's just a single byte — you typically use *hex notation* when writing binary figures. The *hex* is short for *hexadecimal*, which means base 16. Oh boy, yet another base! (Once you're used to working in binary, you'll flip back and forth between hex and binary in your head; converting to decimal is the awkward one.) Recall that in base 2 you have two digits: 0 and 1. In base 10 you have the familiar

digits 0 through 9, but you're not really prepared to write 16 digits. Hex notation borrows the letters A through F for the numbers after 9 that you don't have any digits for: A being 10 and F being 15. Of course, 16 is 10 in hexadecimal. In code, you write a 0x preceding the number to indicate that it's hexadecimal, and you'll use it here to prevent confusion. Now that every place is a factor of 16, 0x1 is 1, 0x10 is 16, and 0x100 is 256. This means that you can write a whole byte in just two characters. Eight bits gets you from 0 up to 255, which can be written in hex from 0x00 to 0xFF. That's why, when you wrote "HI" it was 0x484900 instead of 100100001001001000000000. You can quickly look at a hexadecimal number and determine how many bytes it represents: each two hex digits is 1 byte. So you can quickly tell that this number is 3 bytes, because it has three sets of two numbers. Mentally, you break this number into 0x 48 49 00, and you can tell that the last byte is a null byte — all the bits are 0 — and it's probably a string terminator.

Tip

You can convert numbers to hex in ActionScript by using the numeric types' `toString()` methods, which take a base as an argument. You can even convert to binary this way.

```
var n:int = 0x48;
trace(n); //72
trace(n.toString(16)); //48
trace(n.toString(2)); //1001000
```

As if things weren't complicated enough, bytes can be ordered in memory differently on different platforms. Hopefully, you won't have to deal with this often — the operating system or the language should handle it. But when you're writing custom code to interpret raw data, well, who knows. The ordering of bytes can be called either *byte order* (makes sense) or *endianness*. That word sounds awkward until you realize that the options are *big endian* and *small endian*, as in "which end is the big end?" When you write something as 0x484900, you assume that the bytes are laid out left to right, so you're spelling "HI<NUL>" and not "<NUL>IH. You can make this assumption with confidence, but if you run into a weird problem that looks like the bytes are backward, now you know what the cause may be.

Binary is also the *lingua franca* of images. A simplistic way to store an image is to write the value of each pixel in succession, from left to right and top to bottom. The concept called *color depth* tells you how much space to use to describe each pixel's color. Of course, taking more space yields more possible colors and allows you to render more subtle shades. In ActionScript 3.0, you usually take 4 bytes to describe a color — in other words, 32 bits per pixel (32bpp). The way you encode each byte is also an arbitrary decision; the typical solution is ARGB or RGBA, which means that you take 1 byte each for red, green, blue, and alpha (the amount of transparency) and put them in these orders. `BitmapData` uses ARGB. This is how you end up with colors like 0xFFFF00FF, or totally opaque purple. The alpha value is 255, as are the red and blue values. Red and blue mix additively to form purple. Blend modes and filters use these values as inputs to equations, which are sometimes astoundingly simple. For example, the additive blend mode adds the color values for the two overlapping pixels, no differently than you'd add any two integers. Bitmap graphics are discussed in depth in Chapter 36, "Programming Bitmap Graphics."

Just now, when considering how to store an image, you conceptualized memory as a long, long string of binary, and you thought creatively about how to store something more complicated and structured in this flat space. That's the essence of binary thinking!

Bit Math and Operators

ActionScript 3.0 provides the standard operators for performing bit math. These are identical in many other languages, but I'll review them here. Bit math can be useful for encoding and decoding, for generating random numbers, for optimizing computations, in cryptography, or just to store a set of options as bit flags.

When using bit math, be sure to use the unsigned integer type `uint`.

Basic Arithmetic

Addition and subtraction are the same in binary as they are in decimal, so when you add two `uint`s, the binary numbers add up just as well. It wouldn't be a very good system if two numbers in binary added up to a different sum than they did in decimal.

```
var a:uint = 0xf;
var b:uint = 0x2;
var c:uint = a + b;
trace("0x" + c.toString(16)); //0x11
trace(a.toString(2) + " + " + b.toString(2) + " = " + c.toString(2));
//1111 + 10 = 10001
```

Bit Shifting

Bit shifting is the process of shifting all the digits in a binary string up or down. This is like when you multiply or divide by factors of 10 in decimal and move the decimal place, adding 0s where necessary. Of course, in binary, shifting the digits multiplies and divides the original value by factors of 2.

To shift, use the binary (as in "having two arguments") operators << and >>. These shift the digits to the left and right, or multiply and divide by factors of 2. The operand on the right determines how many places to shift. You can also use the shift-and-assign operators <<= and >>=, which shift the value and assign it back to the same variable. Example 13-1 shows how this works.

Bit Shift Operators

```
package {
  import com.actionscriptbible.Example;
  public class ch13ex1 extends Example {
    public function ch13ex1() {
      var a:uint = 0x4;
      trace2(a);       //  100
      trace2(a >> 1);  //   10
      trace2(a << 1);  // 1000
      trace16(a << 8); //0x400

      var b:uint = 0xb;
      trace2(b);       // 1011
      b >>= 2;
      trace2(b);       //   10
```

```
    b <<= 2;
    trace2(b);        // 1000
  }
  public function trace2(n:uint):void {
    trace(n.toString(2));
  }
  public function trace16(n:uint):void {
    trace("0x" + n.toString(16));
  }
  }
}
```

Notice that there's no decimal place to shift past when you're dealing with integers in binary. So values shifted to the right, like b here, lose whatever digits are truncated forever.

Bit shifting is useful to create one long binary string out of several bytes. You can shift the digits to the left or right, and then add them, to get them to fit together. This is how, if you know the red, green, and blue components of a color, you can reassemble them into one RGB pixel value. Just adding them up isn't right; the blue value is supposed to occupy the last byte, so this would just fill up the blue bucket with all three values. No, you have to align them to bytes, as shown in Example 13-2.

EXAMPLE 13-2 `http://actionscriptbible.com/ch13/ex2`

Bit-Shifting to Create RGB Values

```
package {
  import com.actionscriptbible.Example;
  public class ch13ex2 extends Example {
    public function ch13ex2() {
      //a nice raspberry color, 0xD41153
      var r:uint = 0xD4;
      var g:uint = 0x11;
      var b:uint = 0x53;
      //of course, you could just type in 0xD40253 to your code,
      //but what if the user input these values? You have to reassemble it.
      trace16(r << 16); //0xd40000
      trace16(g << 8);  //0x001100
      trace16(b);       //0x000053
      var rgb:uint = (r << 16) + (g << 8) + b;
      trace16(rgb); //0xd41153
    }
    public function trace16(n:uint):void {
      trace("0x" + n.toString(16));
    }
  }
}
```

Bitwise Logic

You can operate on the bits of an expression using bitwise logical operators. These go through two binary strings and compare each bit using simple rules. ActionScript includes binary AND, OR, XOR, and the unary NOT. These rules for these operators are listed in Table 13-1.

TABLE 13-1

Bitwise Operators

Operator name	Operator	Rule	Example
AND	&	Produces 1 only if both bits are 1.	1100 & 0110 == 0100
OR	|	Produces 1 if either bit is 1.	1100 | 0110 == 1110
XOR	^	Produces 1 if exactly one of the bits is 1.	1100 ^ 0110 == 1010
NOT	!	Inverts every bit.	! 0110 == 1001

Make sure that you don't mix up the binary and logical operators like & and &&. You should also know that all bitwise operations are commutative: the two operands can be in either order.

Bitwise logic can be used in combination with *bit masks*, sequences of bits crafted to zero in on the bits that you want. To do this, you take advantage of the ways that the bitwise rules interact with 0 and 1. For example, AND a sequence with all 1s, and you get the same sequence back.

```
//if the original bit is 1
1 & 1 == 1
//if the original bit is 0
1 & 0 == 0
//so a whole series AND 1s is the same
101100 & 111111 == 101100
```

Likewise, AND a sequence with 0s, and you get 0s back, because both bits must be 1 to output a 1, and you know that one of them is 0.

```
0 & 1 == 0
0 & 0 == 0
101100 & 000000 == 000000
```

You can keep the bits you want and zero out the rest by applying a bit mask.

```
//say you want the first three bits
101100 & 111000 == 101000
//everything else was set to 0
```

Example 13-3 uses a bit mask to pull just the green value out of an ARGB pixel.

Isolating Values with Bit Masks

```
package {
  import com.actionscriptbible.Example;
  public class ch13ex3 extends Example {
    public function ch13ex3() {
      var rgb:uint = 0xd41153;
      var g:uint = rgb & 0x00ff00; //remember, f is 15 is 1111.
      g >>= 8; //you still have to move the bits down
      trace16(g); //0x11
    }
    public function trace16(n:uint):void {
      trace("0x" + n.toString(16));
    }
  }
}
```

Binary Types in ActionScript

Besides the int and uint classes, which are limited to 4 bytes, ActionScript 3.0 provides several classes that support binary operations. These classes have the added benefit of being designed for binary operations, so they are much more convenient to work with than an int. The most important of these classes is ByteArray, which encapsulates a variable-sized binary string. You can think of this as a chunk of memory to which you have access at the byte level. In addition, there are a number of related classes: URLStream and Socket. These give low-level binary access to an HTTP request in progress, or a socket connection to a server (Chapter 28, "Communicating with Remote Services").

Regardless of the destination of the bytes, however, you can understand how to read and write them — thus, to use all these classes — by understanding the common binary stream interfaces IDataInput and IDataOutput. All the concrete classes mentioned — ByteArray, URLStream, and Socket — implement both of these interfaces. A read-only stream, like a URLStream, would only implement IDataInput. A read-write stream, like Socket, implements both. Quite simply, any time you access binary data, you do so in the same way.

The IDataInput and IDataOutput interfaces declare methods that let you read and write objects to a binary stream. With these methods, you can let Flash Player do the work of encoding and decoding various types of data to binary, rather than writing each bit yourself. These methods come in sets, shown in Table 13-2.

TABLE 13-2

Methods of IDataInput and IDataOutput

IDataInput	IDataOutput	Notes
`readBoolean()`	`writeBoolean()`	Read or write a single byte; `0x00` is `false` and any other value is `true`.
`readByte()`	`writeByte()`	Read or write a single byte as an `int`.
`readUnsignedByte()`	`writeUnsignedByte()`	Read or write a single byte as a `uint`.
`readShort()`	`writeShort()`	Read or write a short integer (16 bits) as an `int`.
`readUnsignedShort()`	`writeUnsignedShort()`	Read or write a short integer (16 bits) as a `uint`.
`readInt()`	`writeInt()`	Read or write an integer (32 bits) as an `int`.
`readUnsignedInt()`	`writeUnsignedInt()`	Read or write an unsigned integer (32 bits) as a `uint`.
`readFloat()`	`writeFloat()`	Read or write a single-precision floating point number (32 bits) as a `Number`.
`readDouble()`	`writeDouble()`	Read or write a double precision floating point number (64 bits) as a `Number`.
`readUTF()`	`writeUTF()`	Read or write a UTF-8 encoded string prefixed by the length of the string as an unsigned short integer so that no delimiters are needed.
`readUTFBytes()`	`writeUTFBytes()`	Read or write a UTF-8 encoded string with no prefix. `readUTFBytes()` takes a length parameter specifying the length of the string to read.
`readMultiByte()`	`writeMultiByte()`	Read or write a string in any string encoding, passed as a parameter, such as `us-ascii` or `iso-8859-1`. The full list of supported character sets can be found in the AS3LR.
`readBytes()`	`writeBytes()`	Read or write raw binary as a `ByteArray`.
`readObject()`	`writeObject()`	Read or write an ActionScript object using AMF encoding. Specify AMF0 or AMF3 using the `objectEncoding` parameter.

As you can see, classes that implement these interfaces can encode and decode binary data for you in a variety of formats, giving you flexibility when composing and decomposing tightly packed data.

Using ByteArray

Because a ByteArray is a straightforward implementation of the IDataInput and IDataOutput interfaces, what you learn about ByteArray applies to the other stream classes as well.

Creating a ByteArray

There are several ways to get a ByteArray. The most obvious is to create a new blank one using the constructor. The constructor takes no arguments.

```
var bytes:ByteArray = new ByteArray();
```

Additionally, throughout the Flash Player API, any method that returns binary data does so as a ByteArray object. For example, the BitmapData class has a getPixels() method, which returns a ByteArray.

Writing Data

You can write data to a byte array in two basic ways: use array-access notation ([]) to write to a specific index, and use the methods of IDataOutput described in Table 13-1. Array-access notation works exactly like writing to a standard array.

```
var bytes:ByteArray = new ByteArray();
bytes[0] = 0xff;
```

Of course, each index in a byte array can store only 1 byte: a value between 0 and 255.

When using IDataOutput methods, a variable number of bytes are written depending on the data and its type.

```
bytes.writeInt(1500000050);
trace(bytes[0], bytes[1], bytes[2], bytes[3]); //89 104 47 50
```

Writing to and reading from a ByteArray occurs at a specific position at the stream, exposed by the property position. After every write using an IDataOutput method, this position is moved up automatically to the end of the write. In this manner, without any manual change to position, the ByteArray appends data to the end transparently.

In Flash Player 10 and later, you can clear out a ByteArray with the clear() method.

Reading Data

You can read from a ByteArray in the same ways you write to it: either using array access notation 1 byte at a time, or using the IDataOutput methods. When reading from a ByteArray, the

position determines where objects will be read starting from, and read operations will automatically advance the position to the end of the last read object.

Two properties let you know where you can read from in the array. length reports, in bytes, the total size of the ByteArray. Similarly, bytesAvailable reports the number of bytes left from position to the end of the ByteArray.

Compressing and Decompressing

In Flash Player 10 and later, you can use the built-in deflate algorithm to compress and decompress binary data. Just fill up the ByteArray with the data to compress and call deflate(), or fill it with compressed data and call inflate(). Note that this isn't sufficient to deal with an archive file like a .zip: even if you use a file that has been compressed with the deflate algorithm, it may have metadata and other nondata sections. deflate() works only on the compressed data.

Version

FP10. Compression is available in Flash Player 10 and later. ■

Common Uses of ByteArrays

ByteArray and other binary stream classes are used throughout Flash Player for various purposes. Any place you need to use binary data, keep your ByteArray close at hand. This section covers a few interesting or utilitarian applications. Look throughout the book for plenty of other places ByteArrays are used.

Loading Images

The Loader class, introduced in Chapter 27, "Networking Basics and Flash Player Security," loads images from the internet, but it also can decode and display image files as ByteArray using its loadBytes() method. This is useful when you are loading an image from a socket as a byte stream, for example, as shown in Example 13-4.

EXAMPLE 13-4   http://actionscriptbible.com/ch13/ex4

Loading Images with ByteArray

```
package {
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.net.*;
  import flash.utils.ByteArray;
  public class ch13ex4 extends Sprite {
    protected var urlLoader:URLLoader;
    public function ch13ex4() {
      urlLoader = new URLLoader();
      urlLoader.dataFormat = URLLoaderDataFormat.BINARY;
```

```
    urlLoader.load(new URLRequest(
      "http://actionscriptbible.com/files/caviar.jpg"));
    urlLoader.addEventListener(Event.COMPLETE, onComplete);
  }
  private function onComplete(event:Event):void {
    var loader:Loader = new Loader();
    var bytes:ByteArray = urlLoader.data;
    loader.loadBytes(bytes);
    addChild(loader);
  }
}
}
```

Copying Objects

When copying objects, you must consider several issues. One of the most common programming errors is assuming that an assignment statement will create a new copy of an object. Consider the following example:

```
var arrayA:Array = new Array("a", "b", "c", "d");
var arrayB:Array = arrayA;
```

This might appear to make a copy of `arrayA`, but all it really does is create a new reference to the same object. To see the implications of this, look at the following:

```
arrayB.push("e", "f", "g", "h");
trace(arrayA.length); //8
```

In the preceding example, you add four more items to `arrayB`, but this actually affects `arrayA` as well. This is because both `arrayA` and `arrayB` point to the same object. If you want to create a copy of the array, you can create a new array and use a `for` statement to copy each item from `arrayA` to `arrayB`, as follows:

```
var arrayA:Array = new Array("a", "b", "c", "d");
var arrayB:Array = new Array();
for (var i:int = 0; i < arrayA.length; i++) {
  arrayB[i] = arrayA[i];
}
arrayB.push("e", "f", "g", "h");
trace(arrayA.length);  // Outputs 4
trace(arrayB.length);  // Outputs 8
```

This technique for copying the arrays works, although it is somewhat tedious. However, it gets unmanageable when you have nested objects. For example, consider copying an array of arrays, or an array of arrays of arrays.

You can use a `ByteArray` to sidestep this problem and clone objects neatly. Simply create a new `ByteArray`, write the object to it, reset the position, and read an object out, as shown in Example 13-5. If the object is serialized correctly, you should have an exact duplicate of the original object, not merely another reference.

EXAMPLE 13-5   `http://actionscriptbible.com/ch13/ex5`

Copying an Array with ByteArray

```
package {
  import com.actionscriptbible.Example;
  import flash.utils.ByteArray;
  public class ch13ex5 extends Example {
    public function ch13ex5() {
      var arrayA:Array = new Array("a", "b", "c", "d");
      var byteArray:ByteArray = new ByteArray();
      byteArray.writeObject(arrayA);
      byteArray.position = 0;
      var arrayB:Array = byteArray.readObject();
      arrayB.push("e", "f", "g", "h");
      trace(arrayA.join(" "));  //a b c d
      trace(arrayB.join(" "));  //a b c d e f g h
    }
  }
}
```

Although this may not appear to be a great savings in lines of code initially, consider that the code to copy an object does not get more complex even as the object you are copying gets more complex.

There is one important consideration when you are copying objects of custom types. Byte arrays write data using AMF, which natively supports standard ActionScript data types, but not custom classes. (AMF is the Action Message Format, a binary protocol discussed in greater depth in Chapter 29, "Storing and Sending Data With SharedObject.") That means that, by default, if you write an object of a custom type, it is written to the byte array correctly, but it is stored as a generic object. Therefore, when you retrieve it from a byte array you cannot cast it back to the correct type. You can fix this by using the `registerClassAlias()` function. The function requires two parameters specifying the identifier you want to use to register the class as well as a reference to the class. See this technique used in Chapter 29, in the section "Storing Custom Classes."

Summary

- Binary is the lowest common denominator for storing and retrieving data on a computer; access to binary data allows unlimited interoperability with files and services.
- Binary data is stored in bits, which are grouped into bytes.
- Hex notation is used to concisely write binary strings.

- Binary strings may be operated on by arithmetical and logical operators.

- Use bitwise `AND` with a bit mask and bit shifting to select a subset of bits.

- Binary streams are enabled by the `IDataInput` and `IDataOutput` interfaces.

- `ByteArray` is a sequence of bytes in memory, `URLStream` gives low-level binary access to an HTTP response, and `Socket` is a binary socket. All three implement `IDataInput` and `IDataOutput`.

- Use either array access to access binary data 1 byte at a time, or the `read[xxx]()` and `write[xxx]()` methods declared by `IDataInput` and `IDataOutput` to read and write larger chunks of data.

Part III

The Display List

Visual Programming with the Display List

T he *display list* is the system Flash Player uses to create and manipulate graphics. Flash is and always has been a visual tool, so the display list is at the heart of Flash Player. You've come a long way in the first two parts and gained a good handle on ActionScript 3.0, but once you get your hands on the display list, you have the power to create anything you can imagine on-screen.

In this chapter, you'll learn what the display list is, how it's structured, what classes it contains, and how to use it to create and manipulate graphics. You'll use this knowledge throughout the remainder of the book and often come back to topics in more depth that are touched on here. So don't worry if you see a topic mentioned briefly here, and feel free to jump ahead if it strikes your fancy. For example, vector graphics and the `Shape` object are mentioned briefly here, but they're covered in their entirety in Chapter 35, "Programming Vector Graphics."

The display list is one area of Flash Player that has been completely redone for ActionScript 3.0. It doesn't bear much resemblance to the way things were done in ActionScript 2.0 and Flash Player 8 and earlier. If you're new to AS3, this is a vital chapter to understand.

FEATURED CLASSES

```
flash.display
  .DisplayObject

flash.display
  .DisplayObjectContainer

flash.display
  .InteractiveObject

flash.display.Shape

flash.display.Sprite

flash.display.MovieClip

flash.display.Stage

flash.display.*

flash.geom.Point

flash.geom.Rectangle
```

Introducing Display Lists and Display Objects

The system to display graphics in Flash Player is generically called the "display list." In reality, the entire on-screen state is represented not in a single list but in a tree. And the branches and leaves of this tree are made up of *display objects*, a class of objects that derive from the base class `DisplayObject`. Of course, all display objects can be displayed on-screen, but they have many more interesting properties. This section examines the structure of the display list and the things all display objects can do.

Structure of the Display List

In simpler display paradigms, all things that the computer needs to draw are assigned a depth. Then, using the *painter's algorithm*, the computer draws them in order from the farthest to the closest. When you're painting with a quick-drying paint or ink, you do the same thing. First you paint the sun, the clouds, the sky. Once that's dried, you can paint the grassy knoll, overlapping the sky without worry. Then you can paint the forest in the distance, and finally the school yard on top of that. By drawing everything in the right order, you can overlap these individual layers rather than trying to perfect the edge between them, a tricky problem for you and even for a computer algorithm.

Flash Player has this kind of depth sorting, but additionally it has the concept of *nesting*. If you've ever used Flash Professional and created a clip inside another clip, you know what nesting is. Nesting gives the display list its hierarchy. If you want to paint a giant robot on top of the school grounds, you might use a display object that is nested. Consider the robot a group with a head, torso, and four jointed robotic limbs. By moving these child display objects, you can animate the robot, while the robot as a whole remains a single display object (one composed of several other display objects). When you move the robot, its limbs and head move along with it. How about a more practical example? A dialog box might consist of a background, a prompt, and some buttons (which themselves may consist of a background, text, and so on). The entire dialog box is one display object, but it has many children. When you move the dialog box around the screen, its contents move in lockstep.

Combine nesting and depth, and you have Flash Player's display list model. If a display object has children, all its children have depths. The background of a dialog should appear below its text, or you won't be able to read it. Each display object that has children has a list of those children in order of their depth; this is where *display list* comes from. Take a look at the hierarchy and explicit depth ordering in Figure 14-1.

At the root of the display list in Flash Player is the stage. This is the stage upon which your scene is set, the canvas upon which everything is drawn. There is only one stage, and your whole application must share it. The stage is represented by an instance of the `Stage` class.

Note

The AIR runtime can have multiple stages because every top-level window has its own display list. ∎

The terms `root` and *document class* also relate to objects at the top of the display list, but for now you'll limit yourself to the stage. You can develop applications without explicitly using either of these.

Coordinate Spaces

Every display object owns a *coordinate space*, a frame of reference against which the contents of the display object are measured. Positions, rotations, and distances in ActionScript are expressed in pure numbers. Of course, these naked numbers are meaningless without units like pixels (distance) or degrees (rotation), but they are also meaningless without a coordinate space to exist in. The coordinate space tells you what the *x* and *y* axes are, where those axes meet to form the origin (0, 0), and how far one unit is.
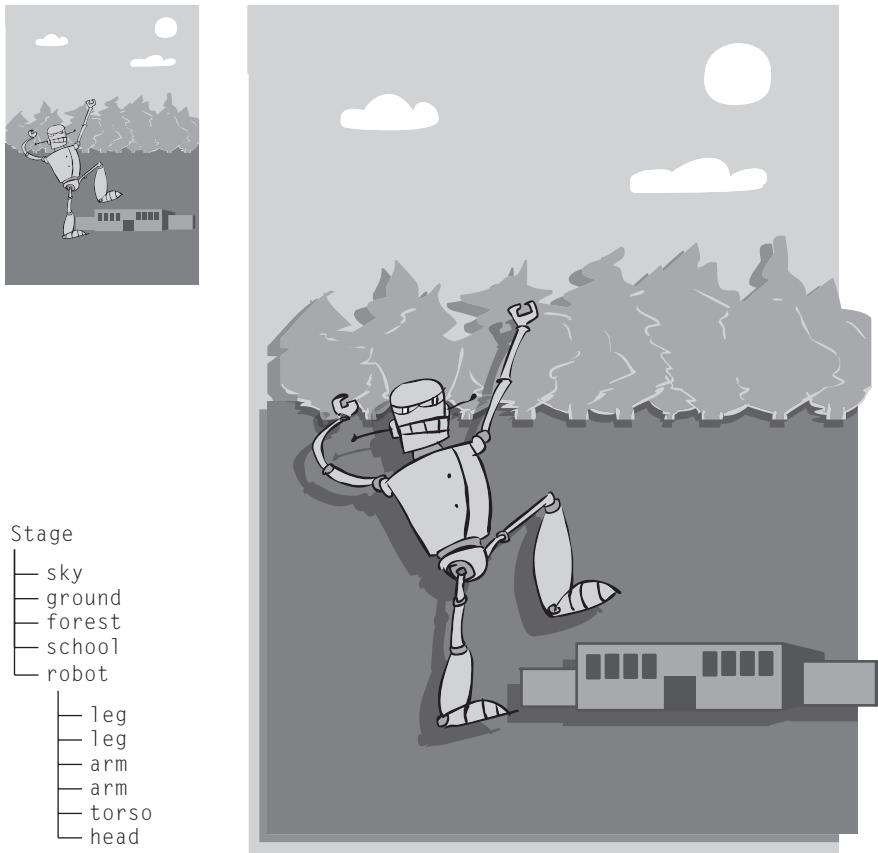
Even though it's easy to forget about it, all measurements are relative to some frame of reference or other. When you measure the height of a building, it's understood that you're speaking about the height directly up starting at the ground — you could just as easily measure the height of the building from sea level instead and get a much different result. When I tell you to go north in Manhattan, I'm actually asking you to travel along the avenues' north-south axis, which is nowhere

near true north — 29 degrees to the east, in fact — and yet perfectly describes the street grid, making it a much more useful frame of reference. Or simply think of it like this: the salt shaker to my right is on your left when you sit across from me. Measurements are always relative to a frame of reference.

A robot rampages the scenic school yard and its display list


```
Stage
 ├── sky
 ├── ground
 ├── forest
 ├── school
 └── robot

         ├── leg
         ├── leg
         ├── arm
         ├── arm
         ├── torso
         └── head
```

The coordinate system of each display object is its *local* coordinate system. If you've created a movie clip in Flash Professional and edited its contents by opening the symbol in the library, you've worked in the symbol's local coordinate system. Suddenly, whatever rotations, translations, scaling, and shearing (collectively known as *transformations*) you've applied to any given instance of the symbol doesn't matter, and you're presented with the clip in its own context. It has an origin — the registration point — from which the *x* and *y* axes emanate to the right and down. In the local coordinate space, one unit is one pixel, and *y* coordinates increase as they go down, unlike the canonical Cartesian coordinate space with *y* increasing "up." In short, the local coordinate system is the natural space of a display object.

Note

Although the pixel is the basic unit of measurement in Flash, space is continuous, and any measurement can be subdivided. Any display object can be placed at any fraction of a pixel. The rendering engine in Flash Player decides how to rasterize fractional pixels depending on certain quality settings. It's not a bad idea to ensure nonrotated bitmaps exist on whole pixels if you want them to remain crisp. ∎

Once you place this display object inside another, its space is subject to whatever transformations you apply to it. Perhaps you have a clip called `tri`, which has a triangle with its top-left corner at (2, 2). This `tri` clip is shown in Figure 14-2. Maybe you add this clip to the stage. Move the clip 5 pixels to the right, for instance, and what once appeared at (2, 2) is at (5, 2), *relative to the stage's coordinate system*. In `tri`'s local coordinate system, nothing has moved or changed, nor will it without your direct intervention. Try to think of these as two independent spaces: `tri`'s coordinate system, and the stage's coordinate system.

Now, the coordinate system that the stage defines is not as flexible. If you forget for a moment the stage scale modes and alignment that you can do when resizing the Flash Player embed or window, the stage is really fixed. You always have this ultimate frame of reference, the *global coordinate system*.

So when you're thinking about where something is on-screen, remember to ask "in relationship to what?" In most cases, it means in relation to the parent coordinate system. In Figure 14-2, you'd say that the triangle inside the display object `tri` is at (2, 2), and the display object `tri` is at (5, 0). You'll see more about moving between frames of reference when you look at the geometry classes later in this chapter. And Chapter 34, "Geometric and Color Transformations," goes deeper into how coordinate spaces may be transformed.

Manipulating the Display List

The display list is mutable — it can be changed. In ActionScript 3.0, you have full access to every object on the display list. Not only can you add in display objects, you can rearrange them, and of course remove them.

Now's a good time to look at some code and see how simple these operations are in ActionScript.

Creating a New Display Object

Creating and instantiating display objects is identical to any other class: use `new`.

```
var carrot:Sprite = new Sprite();
```
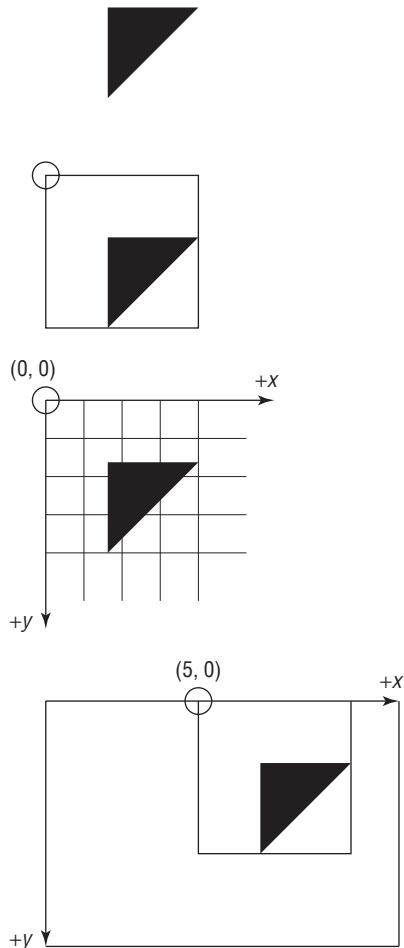
This and the following examples use the `Sprite` display object. You'll spend plenty of time on `Sprite` soon, but for now just know that it's a kind of display object that can have children.

The carrot sprite now exists, but you can't see it. To appear on-screen, a display object must belong to a display list that is on the stage. In other words, you must be able to trace up the tree of display objects all the way to the stage. Conversely, removing a display object from the display list is a simple way to make it invisible, but more on that later.

FIGURE 14-2

A display object, its outline and registration point, its full coordinate system, and its position when added to the stage and translated

Adding an Object to a Display List

You can add a display object to any other display object that can have children. You can pick a depth that this display object will have at the time you add it:

```
stage.addChildAt(carrot, 4);  //assuming there are at least 3 children already
```

Or you can simply let it occupy the next available depth:

```
stage.addChild(carrot);
```

Because carrot is a `Sprite` and it can have children, you can even add children to carrot:

```
var sprouts:Sprite = new Sprite();
carrot.addChild(sprouts);
```

Removing an Object from a Display List

Removing a child is just as easy as adding one.

```
stage.removeChild(carrot);
```

Just as you have to call `addChild()` on the parent of the object to add (you added `carrot` to `stage`), you must call `removeChild()` on the parent of the object to remove.

The display list is just the visual system in Flash Player. Removing a display object from the display list stops it from being drawn but doesn't actually destroy the object or free that memory. The `carrot` variable is still accessible wherever it is normally in scope.

Re-sorting Depths

Depths in the display list have the following properties:

- Depths are indexed — The depth of a display object in a display list is an integer value.

- Depths are unique — Each display object in a display list has a unique depth. No two display objects can coexist at the same depth with the same parent.

- Depths are gapless — It's not possible to leave empty depths in a display list. All contiguous depth values must be occupied.

- Depths increase from the bottom to the top — The "lowest" depth is always zero. A display object at depth zero appears at the bottom, below all overlapping sibling display objects.

Because of these rules, some shuffling of depths already happens when you use `addChildAt()` to insert a display object at a specific depth. When you add a child at an occupied depth, the previous occupant of that depth, and all its higher-depth siblings, are shifted up one to make room for the inserted display object.

It's simple to manually move around sibling display objects within a display list. You can do this by setting the display object's depth directly:

```
var carrot:Sprite = new Sprite();
stage.addChild(carrot);
var apple:Sprite = new Sprite();
stage.addChild(apple); //now you have two children to swap around
stage.setChildIndex(apple, 0); //moves the apple below the carrot
stage.setChildIndex(carrot, 0); //moves the carrot below the apple
```

To follow the depth rules, this method also shifts depths of existing children up to make room for a display object you're moving into an occupied spot. Also following the depth rules, if you try to assign a display object an invalid depth, such as a negative number, or a number higher than the first available depth (which would leave a gap), Flash Player emits a runtime error. Likewise, the first argument to `setChildIndex()` must be a child of the target (here, `stage`), or an error is thrown.

You can also swap two sibling display objects, leaving the rest of the depths intact. There are two ways to accomplish this: swap two specific display objects, or swap whatever display objects happen to occupy the two depths you provide.

```
stage.setChildIndex(apple, 0);
stage.setChildIndex(carrot, 1);
//now carrot is definitively on top of apple
stage.swapChildren(apple, carrot); //now apple is on top
stage.swapChildren(apple, carrot); //now carrot is back on top
//note that swapping again puts things the way they were
stage.swapChildrenAt(0, 1);
//I don't know what was on top before, but now it's on the bottom
```

It should be no surprise that here, too, the inputs must be valid children of the target, and all depths must be valid.

Reparenting Display Objects

You can remove a child from one display object and add it to another. You've already learned all that's necessary to do so. Here you'll find out how an apple looks with carrot sprouts:

```
carrot.removeChild(sprouts);
apple.addChild(sprouts);
```

Examining Display Lists

You can investigate the contents and organization of a display list. You can test whether one display object contains another:

```
carrot.addChild(sprouts);
trace(carrot.contains(sprouts)); //true
```

You can find the depth of a display object:

```
trace(carrot.getChildIndex(sprouts)); //0
```

And you can retrieve child display objects by their depth or name:

```
carrot.name = "Beta Carrotine";
trace(stage.getChildByName("Beta Carrotine")); //[object Sprite]
trace(carrot.getChildAt(0) == sprouts); //true
```

Any display object can have a name, stored in the name property. Most developers only give display objects names when there is a specific reason to, instead manipulating the objects through their references. Named display objects can be useful, though, when displaying graphics created in Flash Professional.
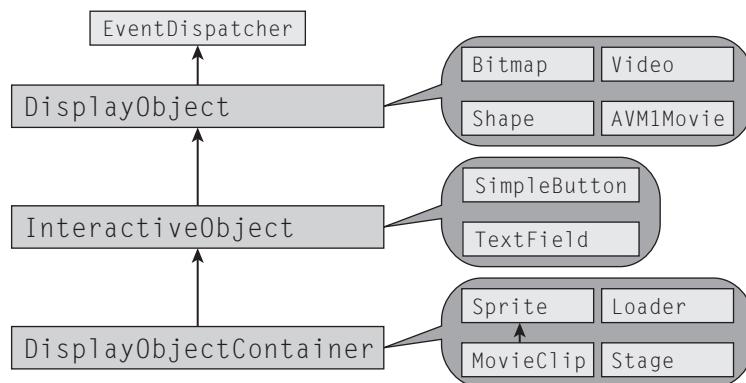
Display Object Classes

There are multiple kinds of display objects, suited for different tasks, all deriving from `DisplayObject`. This section looks at the purpose and specific abilities of these classes and how they relate to each other. Figure 14-3 provides a class diagram of the display objects.

FIGURE 14-3

The display object classes

DisplayObject

The base class for all display objects, `DisplayObject`, provides the properties, methods, and events that are common to all visual objects.

- Position and orientation — Through the properties `x`, `y`, and `rotation`. The coordinate system of Flash Player puts the point (0, 0) at the top-left of the stage, with the positive x-axis pointing to the right, and the positive y-axis pointing down. Rotation is measured clockwise in degrees, and the `rotation` property accepts positive and negative numbers in any range. Each display object has its own coordinate system, and position and orientation are always defined relative to the parent coordinate system. More on this later in this chapter and in Chapter 34.

- Scale — Through the properties `scaleX` and `scaleY`. Scaling lets you stretch an object in one or both directions. Scale is defined in terms of coefficients of the natural size of the object: 0 makes the object invisibly small, 1 is the natural size, 0.5 is half size, 2.0 is double size, and so on. This, like all transformations, is relative to the parent's coordinate system.

- Dimensions — The `height` and `width` properties define the display object's dimensions in pixels. You can resize the object by changing these values; changes here affect the scale as well. There are more precise ways of getting information about a display object's position and size, especially in relation to other coordinate spaces than its parent's.

- 3D properties — In Flash Player 10 and later, display objects may exist in 3D space. In this case, the `z` and `scaleZ` properties are added; and `rotationX`, `rotationY`, and `rotationZ` properties are used instead of a single `rotation`. Chapter 15, "Working in Three Dimensions," dives deep into 3D.

- Opacity — Controlled by the `alpha` property. Values are in the range 0 through 1, where 0 is completely transparent and 1 is totally opaque. Has a cumulative effect on child display objects.

- Visibility — Through the Boolean `visible` property. If set to `true`, the display object (and all its children) is visible.

- Name — As mentioned, display objects may be named with the `name` property.

- Blend mode — Flash Player draws a display object on top of the object it's overlapping following rules set out by the blend mode, set by a display object's `blendMode` property, a string that can be set to any of the static properties in the `BlendMode` class. Flash Player looks at every pixel on the display object and chooses an output color based on the color of that pixel of the display object and the color of the pixel underneath it (which may itself have been created by several other display objects blending together). For example, `BlendMode.ADD` outputs the sum of the color of the display object and everything behind it. This replicates additive colors like light; adding many layers quickly brightens the cumulative color to pure white. These blend modes might be familiar from Photoshop or other image-editing programs. A figure would illustrate this concept well, but not in a black-and-white book. See the AS3LR for a list of available blend modes and examples of them in use. Of special note are `BlendMode.ALPHA` and `BlendMode.ERASE`, which use the alpha channel of the display object to mask or reveal the objects below. These act slightly different from a mask and may in some cases be an easier alternative to what would be a complex mask.

- Masking — Every display object may be masked by another object. The best way to understand this is to look at an example (see Figure 14-4). The visible areas of the mask define which parts of the display object appear; the mask "cuts out" the display object. Set a mask by assigning it to the display object's `mask` property (the mask itself is not visible while it is being used as a mask). Stop masking by setting `mask` to `null`.

FIGURE 14-4

Masking a display object

- Filters — Any number of filters may be applied to any display object. These are defined by the `filters` property, an `Array` of filter objects. The filters are applied in the order found in the array. Filters are covered in depth in Chapter 37, "Applying Filters."

- Scale-9 — Display objects that have vector shapes can be configured to scale like a frame around a picture: the center scales normally, the top and bottom of the frame scale only horizontally, the left and right sides of the frame scale only vertically, and the corners of the frame stay the same size. This lets you design custom edges that distort predictably as the display object is scaled, for example, to retain nicely shaped corners of a button. You can set

this up by assigning a `Rectangle` describing the center cut to the `scale9Grid` property of a `DisplayObject`. Learn more about `Rectangle` in the section "Geometry Classes."

● References — `DisplayObject` defines references to the display object's `stage`, `parent`, and `root`. These are all read-only and may be undefined if the display object is not in the display list of these associated objects.

● Mouse position — The `mouseX` and `mouseY` read-only properties give access to the position of the mouse cursor in relation to the display object, regardless of whether this display object is interactive or has mouse listeners attached.

● Intersection testing — You can see if two objects overlap by passing the second object to the first object's `hitTestObject()` method. Or you can test whether a point falls within a display object with the `hitTestPoint()` method.

● Coordinate space conversion — The display object's transformation matrix is available through the `transform` property. You can transform points from the display object's own coordinate space to the global (stage) coordinate space and back with the `globalToLocal()` and `localToGlobal()` methods and their 3D counterparts. Examine how the display object appears in other coordinate spaces by projecting it with the `getBounds()` and `getRect()` methods.

● Added and removed events — All display objects broadcast `Event.ADDED_TO_STAGE` and `Event.REMOVED_FROM_STAGE` events when added and removed to the stage or any display list on the stage.

● Timing — All display objects broadcast `Event.ENTER_FRAME` events before every frame is drawn, making them ideal timing beacons for animation. See more about time-based events in Chapter 22, "Timers and Time-Driven Programming."

● Drawable — `DisplayObject` implements `IBitmapDrawable`, which means that the contents of any display object may be captured as a bitmap, to be manipulated however you wish. See how to do this in Chapter 36, "Programming Bitmap Graphics."

Phew! This is a huge list of capabilities. What that means for you as a developer is that all display objects are quite capable, which gives you the freedom to create interesting visual expressions. The fact that many of these capabilities are not reserved for specific kinds of display object but are available to all enables some of Flash's most interesting features. For example, text may be drawn in 3D, videos may be filtered, and vector animations may blend with their backgrounds using special blend modes, and other such abominations are possible. Hey, you might even be able to do something pretty, as well.

Now that I've gone to the trouble of describing it, I have to tell you that `DisplayObject` is an abstract class. You can't instantiate it. Although every display object you work with will be derived from `DisplayObject`, not a single one will actually be a `DisplayObject`! Not to worry — they will be even more talented. `DisplayObject` is important because it defines the behaviors that every single display object shares. And in the object oriented world of ActionScript 3.0, having a single base class for all display objects lets the API — and code you write — use display objects polymorphically. Many methods that use display objects accept parameters of type `DisplayObject`, although you will always pass in a subclass of `DisplayObject` instead. This way, any display class is an acceptable parameter.

Instead of `DisplayObject`, use a concrete subclass like `Bitmap`, `Shape`, or `Video`.

InteractiveObject

Classes that derive from `DisplayObject` progressively add on related groups of functionality. The first of these descendants is `InteractiveObject`. An `InteractiveObject` is a display object that reacts to user input from the mouse and keyboard. It adds to `DisplayObject` events and properties useful for an interactive object:

- `mouseEnabled`, `tabEnabled`, `doubleClickEnabled` — Boolean properties that allow certain kinds of interactions on this display object.
- `tabIndex`, `focusRect` — Properties defining tab order and the keyboard focus style, for keyboard-enabled interaction.
- `contextMenu` — Allows you to define a custom context menu for the display object when it is right-clicked.
- `MouseEvent.CLICK`, `MouseEvent.DOUBLE_CLICK`, `MouseEvent.MOUSE_MOVE`, `MouseEvent.MOUSE_OVER`, `MouseEvent.MOUSE_OUT`, `MouseEvent.MOUSE_DOWN`, `MouseEvent.MOUSE_UP`, `MouseEvent.MOUSE_WHEEL` — Mouse events.
- `KeyboardEvent.KEY_DOWN`, `KeyboardEvent.KEY_UP` — Keyboard events.
- `FocusEvent.FOCUS_OUT`, `FocusEvent.FOCUS_IN`, `FocusEvent.MOUSE_FOCUS_CHANGE`, `FocusEvent.KEY_FOCUS_CHANGE` — Focus events.
- `Event.CUT`, `Event.COPY`, `Event.PASTE`, `Event.SELECT_ALL`, `Event.CLEAR` — Interactive text context menu events, available in Flash Player 10 and later.
- `TextEvent.TEXT_INPUT` — When text is input.

These properties and events are covered in Chapter 21, "Interactivity with Mouse and Keyboard Events." Needless to say, when combined they make it possible to interact with the user. `InteractiveObject` is also an abstract class. Instead of using it, you would use a concrete subclass like `SimpleButton` or `TextField`.

DisplayObjectContainer

`DisplayObjectContainer` extends `InteractiveObject`. A basic `DisplayObject` can't contain nested display objects. `DisplayObjectContainer` adds this capability. It adds the methods used in "Introducing Display Lists and Display Objects" to add, remove, reorder, and find children, including:

- `addChild()`, `addChildAt()` — Adding child display objects.
- `removeChild()`, `removeChildAt()` — Removing child display objects.
- `contains` — Testing for parentship.
- `getChildAt()`, `getChildByName()` — Retrieving child display objects by depth or name.
- `setChildIndex()`, `getChildIndex()`, `swapChildren()`, `swapChildrenAt()` — Manipulating child depths.
- `numChildren` — Property (read-only) reporting the total number of child display objects.
- `mouseChildren`, `tabChildren` — Allows or denies certain interactive events to pass through to children. Covered in Chapter 21.
- `getObjectsUnderPoint()` — Retrieves display objects that intersect a given point. Useful for hit detection.

Only use `DisplayObjectContainer` classes when you need the ability to nest child display objects. `DisplayObjectContainer` is also an abstract class. Use its concrete subclasses like `Sprite`, `Loader`, or `TextLine` instead.

Shape

Perhaps the most lightweight display class, `Shape` is a class used for drawing vector shapes. It extends `DisplayObject`, so it is not interactive and cannot nest children. It provides a `Graphics` object to draw programmatically into. `Shape` is covered in Chapter 35.

Bitmap

`Bitmap` is a lightweight class that displays bitmap data, like data loaded from an image file, captured from an `IBitmapDrawable` source, or generated programmatically in a `BitmapData` instance. Most of the interesting bitmap manipulation is done by the `BitmapData` class instead. `Bitmap` holds a reference to a `BitpmapData` object, providing it a means to be displayed in the display list. It extends `DisplayObject`, so it is not interactive and cannot nest children. `Bitmap` is covered in Chapter 36.

Video

The `Video` class provides video content a pathway to the display list. Like `Bitmap` acts as a container for `BitmapData`, `Video` is mostly a container to which you attach a `NetConnection` or a `Camera`. It extends `DisplayObject`, so it is not interactive and cannot nest children. `Video` is covered in Chapter 32, "Playing Video."

AVM1Movie

The `AVM1Movie` class wraps SWFs that were compiled for the AVM1, using ActionScript 1.0 or 2.0, or targeting Flash Player versions 8 and lower. You can place AVM1 SWFs inside your ActionScript 3.0, AVM2 content, and they will display however and wherever you place the `AVM1Movie` display object. The `AVM1Movie` class can't be instantiated by user code; one is automatically created by a `Loader` when an AVM1 SWF is loaded. Although they remain interactive in the same ways that the original SWF was interactive, you can't interface with their code or movie clips; `AVM1Movie` extends `DisplayObject`, so as far as ActionScript 3.0 is concerned, it has no children and can't interact.

SimpleButton

`SimpleButton` is, unsurprisingly, Flash Player's simple button class. As an `InteractiveObject` subclass, it supports interactive events, but it can't contain children. However, it can use up to four `DisplayObjects` as the different graphical states of a button. These states are identical to the states you define on the timeline in Flash Professional for a Button symbol.

- `upState`, `downState`, `overState`, `hitTestState` — The four `DisplayObjects` used as button states. The `SimpleButton` switches between them seamlessly as the mouse interacts with it, showing the `upState` normally, the `overState` while the mouse is hovered over the button, and the `downState` while the mouse button is depressed over the button. The

hitTestState is a display object that defines the active area of the button. It won't be shown, but wherever it is filled, the mouse triggers the button. If omitted, the shape of the current state is used instead, which usually suffices.

If you want a button that reacts with the hand cursor, you can use a SimpleButton, or alternatively, use a Sprite with buttonMode set to true. This is shown in Chapter 21.

TextField

The core class for putting text on the screen — without using the Flash Text Engine — is TextField. It is a subclass of InteractiveObject, so it can't contain children. It defines a host of properties and methods for displaying text, which are covered in Chapter 17, "Text, Styles, and Fonts."

Sprite

One of the true workhorses of the display list, Sprite objects are used ceaselessly in examples in this book. Sprite is the concrete implementation of DisplayObjectContainer; it is interactive and can contain children. Furthermore, it adds these capabilities:

- graphics — A Graphics object, allowing you to draw vectors into a Sprite just like a Shape. Shown in Chapter 34.
- startDrag(), stopDrag(), dropTarget — Allowing you to easily create drag-and-drop behavior.
- buttonMode, useHandCursor, hitArea — Properties that make the Sprite act more like a button.

The one thing that Sprite doesn't do is contain multiple frames, for timeline animations produced in Flash Professional. For that, there exists another class.

MovieClip

The concrete display class MovieClip actually extends Sprite, so it inherits not just interactive and nesting behavior, but the abilities of a Sprite, listed earlier. It adds to this a timeline and timeline controls:

- currentFrame, framesLoaded, totalFrames — Read-only properties that give you information about the frames.
- currentLabel, currentLabels, currentFrameLabel — Read-only properties that provide information about the frame labels in the MovieClip.
- play(), stop() — Methods that control the playhead.
- nextFrame(), prevFrame(), gotoAndPlay(), gotoAndStop() — Methods that control the playhead by frames or frame labels.
- scenes, currentScene, nextScene(), prevScene() — Properties and methods that access and control the scenes of the animation.

Use a MovieClip when you need to have access to timeline animations. Otherwise, stick with a Sprite. You can, in fact, create library items in Flash Professional that are Sprites instead of MovieClips. Chapter 16, "Working with DisplayObjects in Flash Professional," shows you how.

285

Loader

Covered in Chapter 27, "Networking Basics and Flash Player Security," `Loader` is a display object that can go off on its own and retrieve images and SWFs from the web, displaying them when they are finished loading. It extends `DisplayObjectContainer`, but typically you use only its one child, `content`.

Stage

Possibly the most important concrete display class, every Flash application has an instance of the `Stage` class. `Stage` extends `DisplayObjectContainer`, because of course it needs to be able to host children. Adding a display object to the stage is the only way to get it to appear.

Although `Stage` is a concrete class, you can't create your own instances of it. You must access it through the `stage` property of any `DisplayObject` attached to the stage. Furthermore, `Stage` is special in that many of the properties inherited from `DisplayObjectContainer`, `InteractiveObject`, and `DisplayObject` are not applicable or not mutable, and although they are still defined, they raise an error if you try to set a value to them, such as `x`, `y`, and `rotation`, because the stage is the original coordinate system (all coordinates are relative to the stage, and you're not allowed to change the global frame of reference). For a full list of the inapplicable properties, see the AS3LR.

The stage is special in lots of other ways. The capabilities of `Stage` are too important to gloss over, so I'll cover them in depth below.

Resizing

The size of the stage can determine how to draw the interface of your application. For web sites, it's a common technique to embed a SWF to fill up the browser, leaving the resize logic up to ActionScript to handle browser windows of different sizes. Some elements might stay near the top, some might center vertically, and others may hide or show or scale depending on the available room. The size of the stage is an important property.

Although `Stage` exposes `width` and `height` properties like any `DisplayObject`, these measure the size of the content that appears on stage, not the available size of the stage. This can be a common slipup. Instead, the `stageWidth` and `stageHeight` properties report the size available to Flash Player. If the size of the stage changes, it broadcasts an `Event.RESIZE` event. Learn about events in Chapter 20, "Events and the Event Flow."

If you want to take control of how your interface resizes, make sure that the stage's scale mode is set not to scale. You can set this through embed properties — see Chapter 42, "Deploying Flash on the Web" — or using a property of `Stage` described next.

Changing SWF Properties

Many of the properties set during SWF publishing, or by embed properties, may be set or overridden by the `Stage` object.

- `frameRate` — View or change the application's frame rate (how rapidly the screen attempts to redraw). Yes, you can change the frame rate at runtime!

- `quality` — The quality at which Flash Player renders graphics. This affects anti-aliasing of fonts and vector graphics, smoothing of scaled bitmaps, and video. It also affects CPU utilization and potentially the effective frame rate if the CPU can't render frames at the requested quality

and requested rate. Changes in Flash Player over time affect render quality as well, for example, Flash Player 9.0.115 and later use mipmapping (see `http://bit.ly/using-mipmapping`) to scale down bitmaps with high quality even at lower `quality` settings. Flash Player 10 and later use a new text engine that anti-aliases device fonts. Set `quality` to a constant defined by `StageQuality`, like `LOW`, `MEDIUM`, `HIGH`, or `BEST`. See the AS3LR for details on how these impact rendering. You should aim for good performance while the `quality` is set to `StageQuality.HIGH`, and only resort to dropping quality temporarily when it will be noticed less, like during a fast transition. Of course, you can also use `StageQuality.LOW` as a stylistic choice (see `http://wefail.com/`, Flash remakes of retro games).

- `scaleMode` — How the stage scales as Flash Player resizes. Set to a constant defined by `StageScaleMode`. Set to `StageScaleMode.NO_SCALE` to handle resizing manually in ActionScript. Other values automatically scale the entire stage and all its contents, which rarely produces a good effect.

- `align` — How the stage is anchored as it grows if Flash Player is resized. Important if the `scaleMode` is `StageScaleMode.NO_SCALE`. You can anchor the stage to any of the four corners, or the middle of any of the four sides, using constants defined by `StageAlign`. If you're handling resizes in ActionScript, it's typical to align the stage to `StageAlign.TOP_LEFT`, because that ensures that the origin (0, 0) is always at the top left of Flash Player.

- `wmodeGPU` — In Flash Player 10.0.32 and later, this read-only Boolean flag indicates whether the GPU is performing compositing. This property will be true only if GPU compositing is actively in use: if you've indicated that the GPU wmode should be used in your embed parameters or at SWF publish time, and if it is supported by the hardware Flash Player is running on. When this is true, you may be able to crank up the graphics without dropping frame rate. If you have this graphics-intensive of an application, you would do well to keep track of other performance metrics while you adjust.

Going Full-Screen

The `Stage` object is used to bring Flash Player into full-screen mode. In this mode, all or part of the stage is used to fill the entire screen (or one of the screens on a multiscreen device). The embedding parameter `allowFullScreen` must be set to allow the Flash Player plug-in to go full screen. Full-screen mode can only be activated by user input, for example, as a reaction to a click or key press.

Version

FP9. Full-screen display is supported in Flash Player 9.0.28 and later, except on Linux. Full-screen display in Linux is supported in Flash Player 9.0.115 and later.

In Flash Player 9, no keyboard input is accepted in full-screen mode. In Flash Player 10, tab, spacebar, and the arrow keys are accepted. ∎

Once in full-screen mode, Flash Player displays a nonconfigurable prompt that tells the user how to exit full-screen mode with the Escape key. Keyboard access is limited or prohibited (see Note). Graphics hardware may be used to scale up the stage.

To switch to and from full-screen mode, set the `displayState` parameter of `Stage`. Acceptable values are `StageDisplayState.FULL_SCREEN` and `StageDisplayState.NORMAL`. When the display state changes, `Stage` dispatches a `FullScreenEvent.FULL_SCREEN` event that you can use to redraw the interface for full-screen mode.

Version

FP9. Hardware scaling of a portion of the stage, and reporting of the screen dimensions, is available in Flash Player 9.0.115 and later. ■

By default, when going full screen, the stage is resized to the entire size of the target screen. In Flash Player 9.0.115 and later, the `fullScreenHeight` and `fullScreenWidth` properties can give you this value before full-screen mode is engaged; otherwise, you can see what the `stageWidth` and `stageHeight` are resized to.

In Flash Player 9.0.115 and later, you can also choose to only scale up a portion of the stage. This can result in drastic performance gains, because modern screens can be very large — often larger than your application needs (think 2560×1600 pixels) — and redrawing the stage at full resolution at that size can put a lot of stress on the CPU. Instead, you can choose the dimensions of the active area, and they will be scaled up in hardware to fit the screen. The scaling may be evident, but the smooth frame rate you get back is worth the trade-off. Set the area of the stage to be displayed by setting its `fullScreenSourceRect` to the active area of your stage.

Device Orientation

When used on supported devices with accelerometers, such as mobile phones, Flash Player lets you know how the device is oriented, and gives you control over the screen's orientation. You can use this to write an app that can rotate to suit the user's preferred format, or to force the user into the orientation that works best for your application (for instance, if the device is taller than it is wide, you might want to go into a rotated orientation to play a widescreen movie). Orientation can be changed at any time, so you're not locked into a single orientation, and you can even support upside-down orientations.

Version

FP10.1. Orientation support is available in Flash Player 10.1 and higher, and is only available on some devices, primarily smartphones. ■

Possible orientations are denoted by string constants of the `StageOrientation` class:

- `StageOrientation.DEFAULT`
- `StageOrientation.ROTATED_RIGHT`
- `StageOrientation.ROTATED_LEFT`
- `StageOrientation.UPSIDE_DOWN`
- `StageOrientation.UNDEFINED`

The orientation is `UNDEFINED` when the device can't or hasn't yet reported its orientation.

You can get (but not set) the current orientation of the device and the stage with these `Stage` instance properties:

- `orientation` — The current orientation of the screen.
- `deviceOrientation` — The current orientation of the physical device.

Because the screen's rotation can be set independently of the device's, these two don't need to be the same. However, if your intent is to always match the device's orientation, you can easily do so by setting `autoOrients` to `true`.

You can find out whenever the user rotates the device by subscribing to the `Stage`'s `StageOrientationEvent.ORIENTATION_CHANGING` event. The event object contains properties `beforeOrientation` and `afterOrientation` which tell you how the orientation is changing. Learn more about event handling in Chapter 20.

Set the screen's orientation — whether in response to the user rotating the device or manually — by calling `setOrientation()` on the `Stage`, passing in the new desired orientation.

Event Source and Focus Manager

Interactive events are covered in Chapter 21, where you will see that most interactive events find their way to the stage. So you can use `Stage` to capture all keyboard and mouse events.

The stage is also used to control the focused object, that is, the object that receives untargeted user input like keystrokes. I'll cover how to set and retrieve focus in Chapter 21.

Color Correction

The stage can be used to control whether color correction is applied in Flash Player, ensuring accurate reproduction of colors. I'll cover color correction in Chapter 41, "Globalization, Accessibility, and Color Correction."

Geometry Classes

When working with the display list, you will necessarily encounter a lot of geometry. The Flash Player API provides some geometry classes. The two most common are covered here, and their 3D equivalents and the `Transform` class are covered in Chapter 33.

Point

A `Point` represents a position in 2D space. It has `x` and `y` properties of type `Number`. Flash Player's display list API sometimes uses `x` and `y` parameters and sometimes uses `Point`s, but fortunately both representations of a point are simple. When you use `Point`, however, you can take advantage of its built-in geometry operations.

Creating, modifying, and viewing a `Point` are simple, as Example 14-1 shows.

EXAMPLE 14-1   `http://actionscriptbible.com/ch14/ex1`

Working with Points

```
package {
  import com.actionscriptbible.Example;
  import flash.geom.Point;
  public class ch14ex1 extends Example {
    public function ch14ex1() {
      var p1:Point = new Point();
      trace(p1); //(x=0, y=0)
```

continued

EXAMPLE 14-1   *(continued)*

```
        p1.x = 1;
        p1.y = 1;
        trace(p1); //(x=1, y=1);
        var p2:Point = new Point(4, 3);
        trace(p2); //(x=4, y=3)
        trace(p2.length); //5 (the hypotenuse of a right triangle with edges 3,4)
        var p3:Point = p1.add(p2);
        trace(p3); //(x=5, y=4)
    }
  }
}
```

As you can see, `Point's toString()` method is more useful than most. The example also demonstrates the `length` accessor, which returns the distance between (0, 0) and the point — or the length of the `Point` as a vector, but you'll read about `Point`s as vectors in Chapter 33.

`Point` defines some basic methods to manipulate points:

- `add(p2:Point)`, `subtract(p2:Point)` — Add or subtract two points by adding their x and y values, returning a third `Point`.
- `offset(dx:Number, dy:Number)` — Shifts the `Point` by a specified amount without returning a modified value.
- `equals(p2:Point)` — Tests if two points are equal.

`Point` also includes a few static methods for manipulating `Point` instances:

- `distance(p1:Point, p2:Point)` — Returns the distance between the points p1 and p2.
- `interpolate(p1:Point, p2:Point, t:Number)` — Generates a point between the points, faded between p1 and p2 by the t argument. Returns p1 when t is 0, p2 when t is 1, their exact midpoint when t is 0.5, and so on.

There are additional `Point` methods that can prove useful when manipulating positions.

Rectangle

When defining areas, the `Rectangle` class is indispensable. It defines a rectangle by its width, height, and the position of its top-left corner. However, you can access and modify the rectangle by whichever attributes are most useful at the given moment:
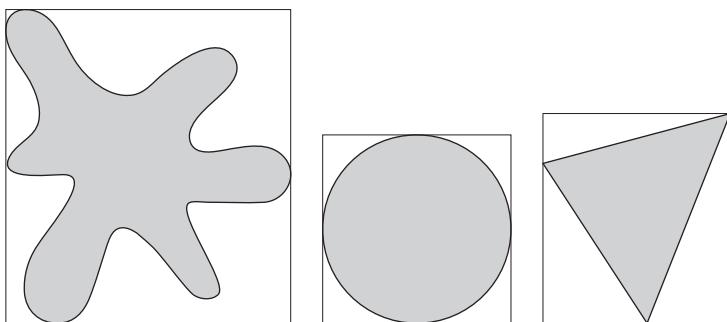
- `left, x` — The x-coordinate of the left edge of the rectangle
- `top, y` — The y-coordinate of the top edge of the rectangle
- `bottom` — The y-coordinate of the bottom edge of the rectangle
- `right` — The x-coordinate of the right edge of the rectangle
- `topLeft, bottomRight` — `Point`s that define the top-left and bottom-right corners of the rectangle

- `width`, `height` — The width and height of the rectangle
- `size` — A `Point` that gives the width and height of the rectangle as (width, height)

Rectangles are used frequently for *bounding boxes*, the smallest axis-aligned rectangle that contains a display object. This representation greatly simplifies what might be a complex shape into something much more flexible, at the cost of accuracy. For example, testing whether two complex shapes overlap mathematically can be difficult, but testing whether their bounding boxes overlap is incredibly simple. Figure 14-5 shows some examples of bounding boxes.

Complex display objects and their bounding boxes

These methods of `Rectangle` provide useful intersection and containment tests:

- `contains(x:Number, y:Number)`, `containsPoint(point:Point)` — Whether the `Rectangle` contains the point specified
- `containsRect(rect:Rectangle)` — Whether the `Rectangle` fully encloses `rect`
- `intersects(rect:Rectangle)` — Whether the `Rectangle` overlaps `rect`
- `equals(rect:Rectangle)` — Whether this `Rectangle` has the same location and dimensions as `rect`

`Rectangle` also supports Boolean geometric operations union and intersection. To look at the bounding box of a set of objects, you might `union()` their bounding boxes. Or to estimate the area that two objects intersect, you might `intersection()` their bounding boxes. These methods return a new `Rectangle` that is the union or intersection of the subject `Rectangle` and the passed `Rectangle`.

Finally, you can shift and inflate the `Rectangle` incrementally. These methods modify the `Rectangle` instance in place and return nothing.

- `offset(dx:Number, dy:Number)`, `offsetPoint(point:Point)` — Shifts the `Rectangle`'s position (without modifying its size) by the specified amount.
- `inflate(dx:Number, dy:Number)`, `inflatePoint(point:Point)` — Increases the size of the `Rectangle` by the specified amount. Keeps the rectangle centered. Its center point does not change, but its position does. Its position is measured from the top-left corner, which moves away from the center, as does the bottom-right corner.

To get the bounding box of a `DisplayObject`, call its `getBounds()` method, passing in the display object you'd like the bounding box to be relative to. Typically, you'll find the bounding box relative to the `DisplayObject`'s parent, because this is how its position, width, height, and rotation are already defined.

```
var carrotBounds:Rectangle = carrot.getBounds(carrot.parent);
```

Or, if the code is running in the context of the `DisplayObjectContainer` that contains the display object you're measuring, you can just pass `this` as the coordinate space:

```
var carrotBounds:Rectangle = carrot.getBounds(this);
```

Sometimes you'll want to see how a deeply nested display object appears relative to the stage in the global coordinate system:

```
var sproutsRect:Rectangle = sprouts.getBounds(stage);
if (sproutsRect.contains(stage.mouseX, stage.mouseY)) {
  trace("you found the sprouts");
}
```

Know that there are better ways to determine if the mouse is over a display object. This merely demonstrates how to use a bounding box. You'll notice that the code compares the mouse position relative to the stage with the sprouts' bounding box relative to the stage. For a point comparison to be meaningful, you have to be talking about two points in the same coordinate system.

Putting the Display List to Use

This chapter has dedicated a lot of space to providing a clear overview of the display list and display objects. But it hasn't been too hands-on with code yet. There's just so much you can do with the display list that an exhaustive set of examples would be impossible. Instead, this chapter focuses on a few common tasks with the display list. In writing interesting examples, I've had to use topics not yet covered, like vector drawing and events, but the focus will be on use of the display list.

Drag-and-Drop, Hit Testing

Let's combine drag-and-drop behavior with hit testing. In Example 14-2, you'll drag visual "files" into a "delete bin," which will remove them from the display list.

EXAMPLE 14-2   `http://actionscriptbible.com/ch14/ex2`

Dragging and Dropping with DisplayList

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.filters.DropShadowFilter;
```

```
public class ch14ex2 extends Sprite {
  protected const NUM_FILES:int = 10;
  protected var deleteBin:Sprite;
  public function ch14ex2() {
    deleteBin = makeDeleteBin();
    //the delete bin should stay at the bottom
    addChildAt(deleteBin, 0);
    deleteBin.x = 15;
    deleteBin.y = 15;

    for (var i:int = 0; i < NUM_FILES; i++) {
      var file:Sprite = makeFile();
      addChild(file);
      //randomize position by looking at available stage size
      file.x = Math.random() * (stage.stageWidth - file.width);
      file.y = Math.random() * (stage.stageHeight - file.height);
      //Sprites are InteractiveObjects
      file.addEventListener(MouseEvent.MOUSE_DOWN, onFileMouseDown);
      file.addEventListener(MouseEvent.MOUSE_UP, onFileMouseUp);
    }
  }
  protected function onFileMouseDown(event:MouseEvent):void {
    var file:Sprite = Sprite(event.target);
    file.startDrag(); //Sprites have simple drag methods
    //moving is relative change in position
    file.x -= 2;
    file.y -= 2;
    //all DisplayObjects support filters
    file.filters = [new DropShadowFilter(2, 45, 0, 0.2)];
    setChildIndex(file, numChildren-1); //set child depth to the top
  }
  protected function onFileMouseUp(event:MouseEvent):void {
    var file:Sprite = Sprite(event.target);
    file.stopDrag(); //Sprites have simple drag methods
    file.x += 2;
    file.y += 2;
    file.filters = [];
    //see if it's over the delete bin
    if (deleteBin.hitTestObject(file)) {
      //and if so, remove from display list
      removeChild(file);
    }
  }
  protected function makeDeleteBin():Sprite {
    var s:Sprite = new Sprite();
    //Sprites support vector drawing
    s.graphics.beginFill(0xff0000);
    s.graphics.drawRoundRect(0, 0, 55, 70, 16);
    s.graphics.endFill();
    return s;
  }
```

continued

293

EXAMPLE 14-2 *(continued)*

```
    protected function makeFile():Sprite {
    var s:Sprite = new Sprite();
    //Sprites support vector drawing
    s.graphics.beginFill(0xc0c0c0);
    s.graphics.lineStyle(0, 0x808080);
    s.graphics.drawRect(0, 0, 8.5, 11);
    s.graphics.endFill();
    s.scaleX = s.scaleY = 3; //DisplayObjects support scaling
    s.buttonMode = true; //Sprites can act like buttons
    return s;
  }
 }
}
```

This example shows not only drag-and-drop and hit testing, but vector drawing, interactive events, filters, and display list adding, removal, and sorting.

Nesting and Cumulative Transformations

In Example 14-3, you'll create a heavily nested display list, in which each link in a chain is the child of the link to its left. When you apply a rotation to only the first link, the chain swivels but remains straight. However, when you apply the same rotation to every link, the chain progressively bends into a circle. This is because each display object is affected by the transformations of its parent.

EXAMPLE 14-3 `http://actionscriptbible.com/ch14/ex3`

Transforms with Nested DisplayLists

```
package {
  import flash.display.CapsStyle;
  import flash.display.Sprite;
  import flash.events.MouseEvent;

  public class ch14ex3 extends Sprite {
    protected const NUM_SEGMENTS:int = 10;
    protected var segmentRotation:Number = 0;
    protected var allSegments:Array;
    public function ch14ex3() {
      allSegments = new Array();
      var segmentLength:Number = stage.stageWidth / NUM_SEGMENTS;
      var segment:Sprite = this;
      segment.y = stage.stageHeight/2;
      for (var i:int = 0; i < NUM_SEGMENTS; i++) {
        var childSegment:Sprite = makeSegment(segmentLength);
        segment.addChild(childSegment);
        childSegment.x = segmentLength;
```

```
        allSegments.push(childSegment);
        //every segment gets added as a child of the last one
        segment = childSegment;
      }
      stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    }
    protected function onMouseMove(event:MouseEvent):void {
      var segmentRotation:Number =
        360/NUM_SEGMENTS * 2*((stage.mouseY / stage.stageHeight) - 0.5);
      for each (var segment:Sprite in allSegments) {
        //all rotation values are set to the same number,
        //yet the line curls inward progressively!
        //this is because every rotation affects all its children.
        segment.rotation = segmentRotation;
      }
    }
    protected function makeSegment(length:Number):Sprite {
      var s:Sprite = new Sprite();
      s.graphics.lineStyle(16, 0x4F7302, 1, false, null, CapsStyle.NONE);
      s.graphics.lineTo(length, 0);
      return s;
    }
  }
}
```

Full-Screen and Stage Resizing

Example 14-4 shows how to resize the interface when the stage size changes. For a drastic change in stage size, the application will go full screen. Remember that the `allowFullScreen` embed parameter must be set to enable full-screen mode on a SWF running in a browser.

EXAMPLE 14-4 `http://actionscriptbible.com/ch14/ex4`

Full-Screen Stage

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.geom.Rectangle;
  import flash.text.TextField;

  public class ch14ex4 extends Sprite {
    protected const KEEP_LAST_N:int = 15;
    protected var fullScreenButton:TextField;
    public function ch14ex4() {
      stage.scaleMode = StageScaleMode.NO_SCALE;
      stage.align = StageAlign.TOP_LEFT;
```

EXAMPLE 14-4 *(continued)*

```
    fullScreenButton = new TextField();
    fullScreenButton.text = "full screen";
    fullScreenButton.selectable = false;
    fullScreenButton.background = true;
    fullScreenButton.backgroundColor = 0xc0c0c0;
    fullScreenButton.border = true;
    fullScreenButton.borderColor = 0;
    fullScreenButton.width = 55;
    fullScreenButton.height = 18;
    fullScreenButton.x = fullScreenButton.y = 25;
    addChild(fullScreenButton);
    fullScreenButton.addEventListener(MouseEvent.CLICK, onFullScreenClick);

    stage.addEventListener(Event.RESIZE, onStageResize);
    onStageResize(null);
  }
  protected function onStageResize(event:Event):void {
    var stageSize:Rectangle =
      new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);
    //create a margin so you can see the indicator
    stageSize.inflate(-10, -10);

    //add an indicator for the screen size
    var stageSizeIndicator:Sprite = new Sprite();
    stageSizeIndicator.graphics.lineStyle(0, Math.random() * 0xffffff);
    stageSizeIndicator.graphics.drawRect(
      stageSize.x, stageSize.y, stageSize.width, stageSize.height);
    var label:TextField = new TextField();
    label.x = stageSize.right - 65;
    label.y = stageSize.bottom - 15;
    label.text = stageSize.width + " x " + stageSize.height;
    stageSizeIndicator.addChild(label);
    addChildAt(stageSizeIndicator, 0);

    if (numChildren > KEEP_LAST_N) {
      removeChildAt(numChildren - 2); //numChildren - 1 is the button.
    }
  }
  protected function onFullScreenClick(event:MouseEvent):void {
    if (stage.displayState == StageDisplayState.NORMAL) {
      stage.displayState = StageDisplayState.FULL_SCREEN;
    } else {
      stage.displayState = StageDisplayState.NORMAL;
    }
  }
 }
}
```

This example also shows simple nesting, addition to and removal from the display list, and text fields.

Rendering and Performance

A great deal of the time spent running your program in Flash Player is used to draw the stage on-screen. Depending on your hardware, version of Flash Player, and embedding parameters, some of the work may be offloaded to a graphics coprocessor, but even so much of the preparation must take place on the CPU. Just some knowledge of Flash Player's rendering pipeline can have a big impact on how fast your Flash application runs, so some basic guidelines are presented here.

Stage Size and Dirty Rectangles

One of the simplest and most obvious rules is that the more area that Flash Player has to redraw, the more time it will take to do so. You should watch out for this especially if you allow your application to be drawn full screen without a `fullScreenSourceRect`.

Flash Player's renderer uses *dirty rectangles* to optimize performance. Flash Player attempts to redraw only portions of the screen that have changed since the last frame. (These are marked as *dirty*.) So a change that affects the whole screen takes much longer to render than a comparable change to a limited area of the screen. For this reason, it's important to remove display objects from the display list when they are not visible. If you fade something out, be sure to set its `visible` property to `false`, or better yet, remove it from the display list entirely. A display object animating while hidden, especially when it is large, needlessly eats up CPU time. You can view the dirty rectangles in debugging builds of Flash Player by choosing Show Redraw Regions in the context menu.

Number of Display Objects

Another simple rule of thumb is that the more display objects you have on stage, the longer a frame takes to render. In reality, the complexity of these display objects determines performance more than their numbers. In particle systems and other applications with high `DisplayObject` counts, culling particles as soon as they become invisible or leave the screen can keep the effects in check. First and foremost, though, make sure the display objects are as simple as possible.

Alpha, Blend Modes, Masking, and Filters

Leaving large areas of a display object filled but transparent forces Flash Player to composite pixels that don't actually appear. Sometimes it can't be avoided, but where it can, you should minimize transparent areas. Also, fully opaque display objects render faster than those with partial transparency using the `alpha` property.

Blend modes should perform comparably overall, except for Pixel Bender blend modes, and the `BlendMode.LAYER` mode, which is discussed later in the section.

Vector masks must be rasterized, so keep them simple. If a mask must be made large to achieve an effect, consider the `BlendMode.REVEAL` blend mode instead.

Filters are computationally expensive and sensitive to the size of the display object being filtered. Thankfully, you can control the quality of most `BitmapFilters`. Make sure you strike a balance between the desired quality and the rendering speed, and when possible, avoid applying filters to large objects. See Chapter 36 for more about filters.

Text

The anti-aliasing type used on dynamic text fields, covered in Chapter 17, has a huge impact on how fast they render. In Flash Player 10 and later, bitmap text is rendered by the host OS and might alleviate some of the load on Flash Player's rendering pipeline, but using bitmap text is not always an option. Be aware that a display object might be costly to render not just if it is text, but if it contains text nested inside it.

Bitmaps, Vectors, and Bitmap Caching

Much of Flash Player's early success is attributable to its ability to render vector graphics well. Vector graphics have the advantage of being highly compressible and retaining quality when they are scaled up. This quality, however, comes at a high cost to the CPU, because you must *rasterize* them, or turn them into pixels, to draw them on the screen. If you compare drawing a bitmap and a vector graphic at their native size and orientation, the bitmap will win in speed, excepting perhaps simple shapes.

The computational expense incurred by rasterizing vector graphics doesn't invalidate their other advantages, however. Flash Player attempts to give you the best of both worlds with an optimization. When you flag a display object to use bitmap caching — by setting its `cacheAsBitmap` property to `true` — Flash Player rasterizes it, including any vectors it may contain, and saves the rasterized bitmap in memory for the next time it needs to draw the display object. The next time, instead of rasterizing the whole thing again, Flash Player can use the cached version, skipping the rasterization. Now, this sounds good and dandy, but there are a number of caveats that boil down to the fact that bitmap caching is only desirable in certain cases, so for the love of all that is good, don't go turning on `cacheAsBitmap` on every `DisplayObject` you see. To wit,

- The cached bitmap must be rasterized afresh any time the display object changes. If this is every frame, bitmap caching actually hurts you.

- The cached bitmap must be rasterized afresh any time the display object is scaled or rotated. Simply scaling or rotating the bitmap won't always produce an image comparable in quality to the vectors rasterized at the new scale and rotation, so Flash Player chooses quality and invalidates the cached bitmap. Simply moving the display object is fine.

- The cached bitmap must live in memory, so using it increases your memory requirements. Keep an eye on this.

Flash Player's bitmap caching only helps if you have large or complex vectors that don't change internally, scale, or rotate. In other cases, using a bitmap representation may still speed up performance a terrific amount, but you have to provide your own bitmap caching method rather than use `cacheAsBitmap`.

A number of operations automatically generate intermediate bitmaps of your display objects. All filters operate on bitmaps. 3D operations in Flash Player 10 and later use a rasterized version of the display object as a 2D texture. And the `BlendMode.LAYER` blend mode flattens the entire display object into a bitmap before compositing it. Use this blend mode when, for example, fading out an entire tree of display objects at once. Instead of applying the inherited alpha value down the display lists, it renders the display object down to a bitmap and applies the alpha value to that instead.

One last word on vectors: gradient fills are slow, and imported artwork may be more complex than necessary. Try optimizing the shapes in Flash Professional or a vector art package.

More on Rendering

While you're just getting started with the display list, don't waste time thinking about how to optimize your graphics. Save that for your first high-performance web site or game. Once you're ready to do serious graphics coding, you'll find a lot of useful information in Part VIII, "Graphics Programming and Animation."

Summary

- The display list is Flash Player's API for compositing graphics.
- Display lists are trees of display objects, with the stage at the root.
- Besides adding display objects to and removing them from a display list, you can change their depth and reparent them.
- Various display classes provide different sets of functionality for different purposes.
- The base display class is `DisplayObject`.
- `InteractiveObject` adds interactivity, and `DisplayObjectContainer` adds the ability to nest children.
- In your own code you must use concrete display classes that extend these base classes, like `Shape`, `Sprite`, and `MovieClip`.
- Flash Player provides geometry classes to simplify geometric programming.

Working in Three Dimensions

F lash Player 10 brings `DisplayObject` into the third dimension, letting you make 3D compositions with just a few lines. Between Flash Player 10's native 3D capabilities and several good software 3D libraries available, 3D has become a fixture of the Flash platform.

In this chapter, you'll learn about Flash Player 10's 3D capabilities and its limitations. You'll also learn the basic way to add 3D effects. There are additional 3D capabilities that support lower-level code and require more math; these will be covered in Chapter 34, "Geometric and Color Transformations," and in Chapter 40, "Advanced 3D." On the other hand, this chapter introduces 3D concepts without too much of the mathematics and theory behind them. You can get plenty of impressive 3D effects with the intuitive API covered here.

I'll also introduce a few widely used ActionScript 3D libraries and compare their features so that you can determine which approach is best for your project, and know where to go for more information.

Version

FP10. This chapter covers features found in Flash Player 10 and later, with the exception of any discussion of ActionScript 3D libraries, which may run in Flash Player 9. ■

FEATURED CLASSES

```
flash.display
  .DisplayObject

flash.display.Stage

flash.geom.Vector3D

flash.geom
  .PerspectiveProjection
```

Introducing 3D in ActionScript 3.0

Up to this point, you've seen the display list and Flash Player's coordinate space as flat; two-dimensional. Display objects can have a position, scale, rotation, and even a stacking order, but they can't escape the fact that they all exist on a flat plane. Of course, when you think of it, your screen is a flat plane, and what you think of as a 3D scene is really just a flat image rendered with the simulation of depth. This trick of rendering is not new, having been incorporated by artists in the 15th century; the trick is called *perspective*. Regardless of its age, Flash Player only adopted this trick in version 10.0. And believe it or not, drawing "in three dimensions" does come down to this one trick.

As an interesting aside, there is one transformation of a display object that I didn't mention earlier: skew. Skew doesn't have its own `DisplayObject` property like the ones I listed, but it can be added in all versions of Flash Player using the display object's transformation matrix (see Chapter 34), or visually in Flash Professional. Given that you can create the illusion of 3D by drawing a scene in a certain way (with a certain *projection*), there's actually another way to draw 3D that relies on skewing. This kind of drawing is a *parallel projection*. You may be familiar with parallel projections if you've seen schematic drawings — blueprints — or played three-quarter, top-down games like *SimCity*, *Final Fantasy Tactics*, *Diablo*, and so on. The problem with parallel projections is that parallel lines never appear to converge.

Parallel projections like these are mathematically pleasing, but you see in perspective where parallel lines eventually converge. (Imagine train tracks going off into the distance. At the horizon, they will have merged into a single point.) Perspective transforms just can't be done with a skew, because as you can see in Figure 15-1, skewing preserves parallel lines. So all it really takes to create the illusion of depth is a perspective transformation. Figure 15-1 shows two rectangles — one skewed, and one with a perspective transform — and the kinds of scenes that can be drawn with those different transformations.

FIGURE 15-1

Skew and perspective transformations; parallel and perspective projections

What all this boils down to is that you need a perspective transformation to convincingly fake three dimensions, and this kind of transformation is only available in Flash Player 10 and later. Now that you know what the trick is, take three giant steps back, because you're not going to have to apply it directly (in this chapter). When you use the third dimension in Flash Player's display list, it applies a perspective projection automatically. You just focus on positioning objects in three-dimensional space; Flash Player draws the convincing 2D perspective projection on-screen for you.
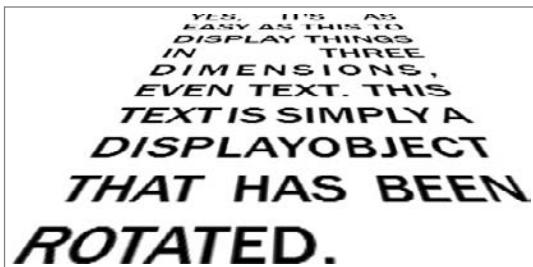
Note

While Flash Player 9 can't do a perspective projection natively, of course, you can write code that approximates one. This four-point distortion is exactly how ActionScript 3D libraries work, in versions before Flash Player 10. ■

This is a critical point: when using 3D in the display list, all you have to do is position and orient display objects in three dimensions. There are really no extra classes or steps required. You'll use `DisplayObject` and all its subclasses in the same way, except with a dimension added. Figure 15-2 is simply a block of TLF text (it could as easily be a `TextField`, `Shape`, or `Video`) rotated about the x-axis and positioned in 3D space, nothing more.

FIGURE 15-2

A DisplayObject rotated in 3D space

The 3D Coordinate System

In Chapter 14, "Visual Programming and the Display List," you saw that the coordinate space that Flash Player uses is a Cartesian space with the origin at the top-left corner of the stage. The x-axis increases to the right and the y-axis increases down. Because of this, points in any quadrant but the bottom-right don't appear on-stage.

When you're using the 3D coordinate system, none of this changes. A third, z-axis, is added. The origin stays at the top-left corner, and the z-axis protrudes perpendicular to the stage, into and out of your display, with positive z being farther away from you, and negative z being closer to you. By default, the perspective is set up so that the vanishing point is in the center of the stage, infinitely far off in positive z. You can change this, but for now, it places your eye squarely in front of the xy plane — the traditional 2D stage — looking into it. Figure 15-3 shows the 2D and 3D coordinate systems of the display list.
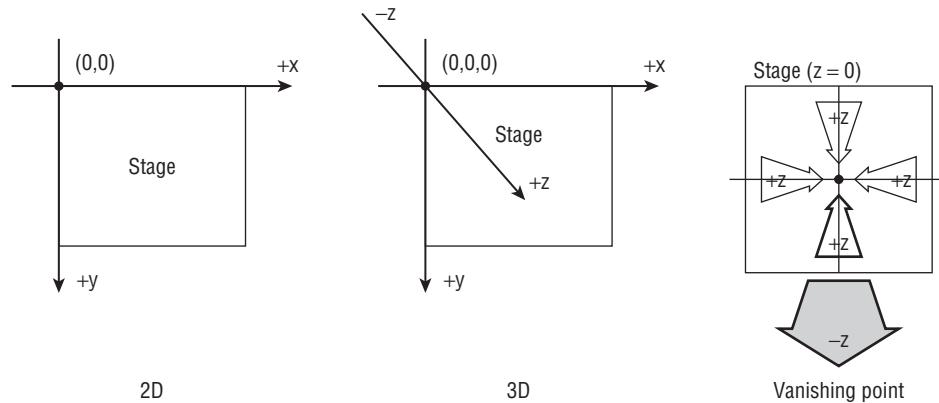
You can think of the stage as existing at z=0. Because z increases "into" the screen, objects with positive z values appear smaller than their natural size — as if they have been scaled down — and objects with negative z values appear larger than their natural size — as if they have been scaled up. In actuality, there is only one stage, and display objects in 2D and 3D must coexist in peace on it. So, those 2D display objects are essentially at z=0.

Display objects can not only be positioned in 3D space, they can be rotated and scaled in three dimensions as well. The text in Figure 15-2 is rotated around the x-axis. Figure 15-4 shows rotations around each of the three axes. You'll notice that because the z-axis points in and out of the screen, rotating around the z-axis is the same as traditional 2D rotation. There are plenty of ways to represent a rotation in 3D space — it's not nearly as simple as rotation in 2D — but the display list represents any orientation as a combination of these three axis rotations (an Euler angle). Although the figure shows an object rotated around a single axis, combining three rotations, one around each axis, can produce any orientation imaginable.

303
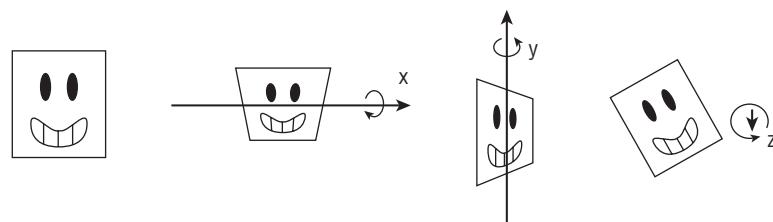
FIGURE 15-3

The 2D and 3D coordinate systems, axes, and origins; direction of +z ''into'' the screen

FIGURE 15-4

Display objects can be rotated in all three dimensions. Rotating around the z-axis is the same as 2D rotation.

3D in Flash Professional

Setting up a scene in three dimensions may be much easier when you can see exactly what goes where. Flash Professional starting in CS4 contains tools to manipulate objects in three dimensions, which may be much easier than trying to guess at the orientation and positions of objects in 3D space. Alternatively, you can lay out a scene in a 3D package and copy down the positions from the tool into code. I won't show how to use those tools here, but know that it can help you visualize things that are difficult to picture from numbers alone.

Limitations of 3D Display Objects

Although it's perfect for simple effects, the 3D capabilities of the display list in Flash Player 10 are far from what you'd expect in a 3D game or modeling tool. Nor should they be comparable. You should know what the display list is capable of so that you know where and when to deploy it.

Display Objects Are Flat

First and foremost, using 3D in the display list is not what most consider a 3D engine, but rather "2.5D" or a "billboard" engine. It only goes as far as projecting `DisplayObjects` in perspective. But all these objects are flat, like the face in Figure 15-4 and the text in Figure 15-2. In Chapter 14 you reviewed all the types of display object. Can you think of a single one that has any thickness? There are none. Display objects are like pieces of paper or billboards. You can view them in perspective, but if you want to show an object with more volume, it's not like you can load in a 3D model and display it in the display list. You could, potentially, construct one like papercraft by creating a display object for each face.

This means that 3D using the display list is well suited to some applications. It could power a game with flat sprites. Before the advent of really good 3D engines and hardware 3D acceleration, games layered flat art in 3D space, and people still had a good time playing games like *Wolfenstein 3D*, *DOOM*, and *Duke Nukem 3D*. All of these games placed flat sprites in a perspective 3D space. You could use it to create a multilayered UI; oddly, most user interfaces are still flat, but 2D elements like windows and panels presented in 3D can be a distinctive experience. Think of iTunes' Cover Flow, the signature "flip" seen in the iPhone OS, or the Xbox 360 Dashboard.

A Viewport Isn't a Camera

In a proper 3D package, you'd hopefully think in terms of scenes, containing models, some lights, and at least one camera. You learned how the display list is organized in Chapter 14, so you'll remember that it's not at all like that. When Flash Player goes through the display list to render a scene, it uses a projection matrix to render the 3D space defined with the origin at the corner of the stage. On the other hand, a camera (like the first-person view in any FPS game) is mobile: it moves around the scene pointing in every direction. Making a 3D application with display list 3D where the camera is anything but fixed is not impossible but would require some feats of abstract thinking. (So you may want to take that *DOOM* clone that I just told you was possible and put it on rails.)

Depths Are Managed by the DisplayObjectContainer

Another implication of the fact that the display list is a display list, not a 3D engine, is the lack of z-sorting. Z-sorting automatically draws stuff that's closer to you (its position has a greater z value) on top of stuff that's farther away. If you're watching a giant robot crushing a schoolyard right in front of you, you'll see the robot overlapping the mountain range in the distance, not the other way around. But the display list, in 2D or 3D, obeys the depths of its children to determine layering order. This is a fatal omission for any 3D engine, because things look horrendously wrong when placed out of order.

You can easily write a simple z-sorting algorithm that looks at the position of each `DisplayObject` to set its depth. An algorithm that works for multiple objects is contributed by Drew Cummins at `http://bit.ly/cummins-z-sort`. This works well, but it can't patch one flaw: display objects have area; they aren't single infinitesimal points. If you only look at the position of the center of an object — especially a long one — you can make incorrect assumptions about its layering.

Other Missing Stuff

Lighting and shading, distance fog, backface culling — you could make a huge list with everything display list 3D doesn't do. And it would contain everything 3D engine *does* do, with the exception of perspective projection. So you get it. It's cool to put billboards in 3D space, but don't expect much. Let's find out how, already.

DisplayObject Revisited

All it takes to make a `DisplayObject` 3D is to move it or orient it in 3D space. The existing properties that control position, rotation, and scale in two dimensions are simply augmented with a third dimension. These properties are listed in Table 15-1.

This table and the properties within shouldn't need much further explanation at this point, save a few notes.

TABLE 15-1

2D and 3D Properties of DisplayObject

2D property	3D property
x	x
y	y
	z
	rotationX
	rotationY
rotation	rotationZ
scaleX	scaleX
scaleY	scaleY
	scaleZ
width	width
height	height

Because `DisplayObjects` have no depth, `scaleZ` does nothing. You'll also note that there's no `depth` added to `width` and `height`. So why have `scaleZ`? Perhaps because the math requires it. It can actually take on any value, which isn't surprising when you realize that the natural depth of any `DisplayObject` is 0, which no scale value will ever change.

The three rotation values are measured in degrees, just like the 2D `rotation` property. Because `rotationZ` is the same kind of rotation in 3D that `rotation` is in 2D, you can still use the `rotation` property in 3D, which sets both `rotation` and `rotationZ`. However, you shouldn't. Use `rotationZ` in 3D and `rotation` in 2D.

All you have to do is set any one of the 3D-only properties, and the object will be rendered in 3D, although geometrically, there's no special distinction between 2D and 3D `DisplayObjects`. 2D `DisplayObjects` simply lie on the plane of the stage. However, to Flash Player, there's a big difference.

To draw a display object in 3D, Flash Player first renders it down to a bitmap at its natural size. This has an effect similar to turning on `cacheAsBitmap`. It also means that objects that display larger than their natural size — ones that are closer to you than the stage, or have negative z coordinates,

that is — may appear blocky or smudgy, like a scaled-up bitmap. You can see this bitmap scaling if you look closely at Figure 15-2.

You'll remember from Chapter 14 that every `DisplayObjectContainer` defines its own coordinate space. This way, you can nest clips and create parent-child relationships. Move the parent, and the child moves along with it, its position relative to the parent unchanged. The same applies to display objects in 3D. In Example 15-1, you'll create a `Cube` display object that contains six faces. Each face will have its own position and rotation that places it in the shape of a cube. Then you'll experiment with placing the entire cube in 3D space. As you'd expect, all the faces of the cube keep their relative positions, and you can move around the whole thing like any `DisplayObject`.

EXAMPLE 15-1 `http://actionscriptbible.com/ch15/ex1`

Creating a 3D Display Object

```
package {
  import flash.display.Sprite;
  public class ch15ex1 extends Sprite {
    public function ch15ex1() {
      var boringCube:Cube = new Cube(100);
      boringCube.x = stage.stageWidth * 0.5;
      boringCube.y = stage.stageHeight * 0.5;
      addChild(boringCube);

      var cube:Cube = new Cube(100);
      cube.rotationX = -40;
      cube.rotationY = 20;
      cube.rotationZ = 12;
      cube.x = stage.stageWidth * 0.2;
      cube.y = stage.stageHeight * 0.6;
      cube.z = 150;
      addChild(cube);

      cube = new Cube(100);
      cube.rotationX = 95;
      cube.rotationY = -30;
      cube.rotationZ = 12;
      cube.x = stage.stageWidth * 0.7;
      cube.y = stage.stageHeight * 0.4;
      cube.z = -150;
      addChild(cube);
    }
  }
}
import flash.display.*;
class Cube extends Sprite {
  public function Cube(size:Number = 50) {
    for (var side:int = 0; side < 6; side++) {
      var face:Sprite = new Sprite();
      face.blendMode = BlendMode.MULTIPLY;
```

continued

307

EXAMPLE 15-1 *(continued)*

```
        var brightness:int = 0x80 + 0x80 * Math.random();
        var color:int = brightness << 16 | brightness << 8 | brightness;
        face.graphics.beginFill(color, 0.8);
        face.graphics.drawRect(-size/2, -size/2, size, size);
        face.graphics.endFill();
        addChild(face);
      }
      getChildAt(0).rotationY = 90; //left
      getChildAt(0).x = -size/2;
      getChildAt(1).rotationY = -90; //right
      getChildAt(1).x = size/2;
      getChildAt(2).z = -size/2; //front
      getChildAt(3).rotationY = 180; //back
      getChildAt(3).z = size/2;
      getChildAt(4).rotationX = 90; //bottom
      getChildAt(4).y = size/2;
      getChildAt(5).rotationX = -90; //top
      getChildAt(5).y = -size/2;
    }
  }
```
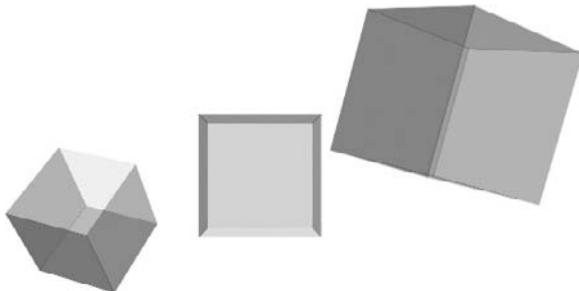
The three cubes are shown in Figure 15-5. The `boringCube` doesn't have a transformation of its own (except repositioning it so that it's visible), to contrast it from the `Cube`s, which have more complicated transformations. All three `Cube` instances hold together because their children are positioned in the parent `Cube` coordinate system, and the parent is what's being rotated and moved. All transformations are also relative to the origin of the coordinate system, which you were careful to place in the center of the cube. This makes the cube rotate around the center, which is far more predictable and easier to program. For more proof, you can easily modify the example by drawing a point at the origin of each `Cube`'s coordinate system, or better yet, add three `Shape`s to show its own three axes.

FIGURE 15-5

Three cubes, each with their own identical children and coordinate system. The parent coordinate systems are transformed in different ways. Displayed when you run Example 15-1.

Another observation you can draw from this example is that the depth sorting is completely dependent on the display list's ordering, not on the object's position in 3D space. I've intentionally used translucent cubes so that you can't tell the incorrect ordering with a glance. Just turn the blend mode back to normal and make the faces opaque, and you'll see that the display list ordering is in effect, including the fact that all faces in one cube are either above or below all faces in another cube, because they're separate `DisplayObjectContainers` with their own child layering.

Geometry Revisited
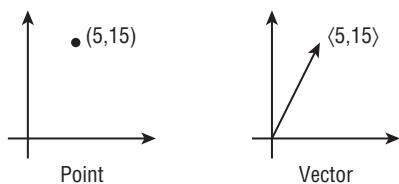
There are new geometry classes in Flash Player 10 for 3D geometry, `Vector3D` and `Matrix3D`, found in the `flash.geom` package. (In the next section, you'll also touch on the `PerspectiveProjection` class in this package.) You'll find both of these thoroughly explained in Chapter 34, but I'll quickly introduce `Vector3D` here.

The `Vector3D` class represents a point in three dimensions, much like `Point` represents one in two dimensions. It contains x, y, and z properties. It also contains a fourth coordinate, w, to be ignored for now. Now, there's nothing that's forcing you to use this class only for coordinates. You can also use it to represent a *vector* in 3D. This *vector* has absolutely nothing to do with the `Vector` class in Chapter 9, "Vectors." A vector in the geometric sense is a direction and a length — you can think of it as an arrow pointing from the origin to (x, y, z), as Figure 15-6 demonstrates. The difference between a vector and a point is subtle. Say you're driving to New York City. A coordinate tells you where it is. A vector points you there. They both have the same values.

FIGURE 15-6

A vector and a point in 2D

For instance, you'd use a vector to encode a distance, a velocity, an acceleration, an Euler rotation, or in Example 15-2, an angular velocity.

EXAMPLE 15-2 `http://actionscriptbible.com/ch15/ex2`

Using Vectors in 3D

```
package {
  import flash.display.*;
  import flash.events.MouseEvent;
  import flash.geom.*;
```

```
  public class ch15ex2 extends Sprite {
    public function ch15ex2() {
      stage.quality = StageQuality.LOW;
      stage.addEventListener(MouseEvent.CLICK, onClick);
    }
    protected function onClick(event:MouseEvent):void {
      var cube:Cube = new Cube(20);
      cube.x = event.localX;
      cube.y = event.localY;
      addChild(cube);
      var normalizedMousePosition2D:Point = new Point(
        stage.mouseX / stage.stageWidth * 2 - 1,
        stage.mouseY / stage.stageHeight * 2 - 1);
      cube.angularVelocity.x = Math.pow(normalizedMousePosition2D.y, 3) * 10;
      cube.angularVelocity.y = Math.pow(normalizedMousePosition2D.x, 3) * 10;
    }
  }
}
import flash.display.*;
import flash.events.Event;
import flash.geom.Vector3D;
class Cube extends Sprite {
  public var angularVelocity:Vector3D;
  public function Cube(size:Number = 50, ang:Vector3D = null) {
    for (var side:int = 0; side < 6; side++) {
      var face:Sprite = new Sprite();
      face.blendMode = BlendMode.MULTIPLY;
      var brightness:int = 0x80 + 0x80 * Math.random();
      var color:int = brightness << 16 | brightness << 8 | brightness;
      face.graphics.beginFill(color, 0.8);
      face.graphics.drawRect(-size/2, -size/2, size, size);
      face.graphics.endFill();
      addChild(face);
    }
    getChildAt(0).rotationY = 90; getChildAt(0).x = -size/2;
    getChildAt(1).rotationY = -90; getChildAt(1).x = size/2;
    getChildAt(2).z = -size/2;
    getChildAt(3).rotationY = 180; getChildAt(3).z = size/2;
    getChildAt(4).rotationX = 90; getChildAt(4).y = size/2;
    getChildAt(5).rotationX = -90; getChildAt(5).y = -size/2;
    this.angularVelocity = (ang)? ang : new Vector3D(0, 0, 0);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  protected function onEnterFrame(event:Event):void {
    this.rotationX += angularVelocity.x;
    this.rotationY += angularVelocity.y;
    this.rotationZ += angularVelocity.z;
  }
}
```

In Example 15-2, you use the x, y, and z properties of a `Vector3D` to represent the rotation to be added to each axis in each frame. Just like a `Point`, these properties can be set when calling the constructor. Don't worry about the event handling and animation here.

`Vector3D` has a wonderful set of methods that do vector math; you may find simple ones like `add()`, `subtract()`, `scaleBy()`, and `distance()` useful here. Once you start delving into projections and matrix multiplication, you'd be better off heading to Chapter 34 to learn about 3D transformation matrices.

Mouse and Point Translation in 3D

The good news is that mouse position is mapped automatically and correctly onto 3D objects. A button in 3D can register a click no matter its angle. The mouse's position is always a 2D point in the display object's coordinate space, and Flash Player projects the mouse position onto a 3D display object without intervention from you.

Although Example 15-3 uses the drawing API and mouse events, which are covered in later chapters, it does offer convincing proof that mouse positions are projected correctly in 3D space.

EXAMPLE 15-3 `http://actionscriptbible.com/ch15/ex3`

Clicking 3D Objects

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  public class ch15ex3 extends Sprite {
    protected var canvas:Canvas;
    public function ch15ex3() {
      canvas = new Canvas(400, 300);
      addChild(canvas);
      canvas.x = stage.stageWidth/2;
      canvas.y = stage.stageHeight/2;
      canvas.rotationY = 25;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      canvas.rotationY += 0.2;
      canvas.rotationX += 0.04;
    }
  }
}
import flash.display.Shape;
import flash.display.Sprite;
import flash.events.MouseEvent;
import flash.geom.Rectangle;
class Canvas extends Sprite {
  protected var fg:Shape, bg:Shape;
```

continued

EXAMPLE 15-3 *(continued)*

```
protected var ink:int = 0;
public function Canvas(w:Number, h:Number) {
  bg = new Shape();
  bg.graphics.beginFill(0xe0e0e0, 1);
  bg.graphics.drawRect(0, 0, w, h);
  bg.x = -w/2; bg.y = -h/2;
  addChild(bg);
  fg = new Shape();
  fg.x = -w/2; fg.y = -h/2;
  addChild(fg);
  fg.scrollRect = new Rectangle(-w/2, -h/2, w, h);
  addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
}
protected function onMouseMove(event:MouseEvent):void {
  if (event.buttonDown) {
    ink++;
    fg.graphics.lineTo(event.localX, event.localY);
  } else {
    if (ink > 500) {
      fg.graphics.clear();
      ink = 0;
    }
    fg.graphics.lineStyle(32, 0xffffff * Math.random(), 0.4);
    fg.graphics.moveTo(event.localX, event.localY);
  }
}
}
```

Run the example, and you'll see a canvas in 3D that you can paint on as it rotates. Now, I wrote the `Canvas` code with no specific concessions for 3D, but it works in 2D or 3D equally well. You'll notice that wherever you place your virtual pen in 3D, it's put on the canvas right below the mouse cursor as if your pen were an infinite line parallel to the z-axis.

Translating Points in Code

In many cases, ActionScript handles local, 2D coordinates and global, 3D coordinates by itself, as in Example 15-3. If need be, though, you can convert between two and three dimensions in ActionScript easily, using these methods of `DisplayObject`:

- `globalToLocal3D()` — Takes a 2D global `Point` — a point on the stage — and finds out where that point is in 3D space relative to the `DisplayObject`. In other words, it finds what `Vector3D` points from the origin of the `DisplayObject` to the point on stage.

- `local3DToGlobal()` — Takes a 3D `Vector3D` point in the local coordinate system and finds out where that point is drawn on stage in 2D.

Example 15-4 will clarify these point projections. You'll track vertices on a rotating cube, using `local3DToGlobal()` to convert the vertex's position to its position on-screen. Then you'll draw on-screen where the vertex is. In this manner, you'll have some of the corners trace out a path as the cube rotates.

EXAMPLE 15-4 `http://actionscriptbible.com/ch15/ex4`

Translating 3D Points to 2D

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.geom.*;
  public class ch15ex4 extends Sprite {
    protected var cube:Cube;
    protected var angularVelocity:Vector3D;
    protected var layers:Vector.<Shape> = new Vector.<Shape>();
    protected var colors:Vector.<uint> = new <uint>[0xD97C2B,0x496B73];
    protected var firstFrame:Boolean = true;
    public function ch15ex4() {
      cube = new Cube(200);
      cube.x = stage.stageWidth/2; cube.y = stage.stageHeight/2;
      addChild(cube);
      for (var i:int = 0; i < colors.length; i++) {
        layers[i] = new Shape();
        addChild(layers[i]);
      }
      stage.addEventListener(MouseEvent.CLICK, onClick);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
      onClick(null);
    }
    protected function onClick(event:MouseEvent):void {
      var r:Function = function():Number {return Math.random() * 8 - 4};
      angularVelocity = new Vector3D(r(), r(), r());
      firstFrame = true;
      for (var i:int = 0; i < layers.length; i++) {
        layers[i].graphics.clear();
        layers[i].graphics.lineStyle(0, colors[i], 1);
      }
    }
    protected function onEnterFrame(event:Event):void {
      cube.rotationX += angularVelocity.x;
      cube.rotationY += angularVelocity.y;
      cube.rotationZ += angularVelocity.z;
      for (var i:int = 0; i < layers.length; i++) {
        var p:Point = cube.local3DToGlobal(cube.vertices[i]);
        if (firstFrame) {
          layers[i].graphics.moveTo(p.x, p.y);
        } else {
```

continued

EXAMPLE 15-4 *(continued)*

```
          layers[i].graphics.lineTo(p.x, p.y);
        }
      }
      firstFrame = false;
    }
  }
}
import flash.display.*;
import flash.geom.Vector3D;
class Cube extends Sprite {
  public var vertices:Vector.<Vector3D>;
  public function Cube(size:Number = 50) {
    var S2:Number = size/2;
    for (var side:int = 0; side < 6; side++) {
      var face:Sprite = new Sprite();
      face.graphics.lineStyle(8, 0, 0.1, false, LineScaleMode.NORMAL);
      face.graphics.drawRect(-S2, -S2, size, size);
      addChild(face);
    }
    vertices = new Vector.<Vector3D>();
    vertices.push(new Vector3D(-S2, -S2, -S2));
    vertices.push(new Vector3D(S2, S2, S2));
    getChildAt(0).rotationY = 90; getChildAt(0).x = -S2;
    getChildAt(1).rotationY = -90; getChildAt(1).x = S2;
    getChildAt(2).z = -S2;
    getChildAt(3).rotationY = 180; getChildAt(3).z = S2;
    getChildAt(4).rotationX = 90; getChildAt(4).y = S2;
    getChildAt(5).rotationX = -90; getChildAt(5).y = -S2;
  }
}
```

You can see the code in action in Figure 15-7. The corners being traced are indicated by arrows.

Modifying the Projection
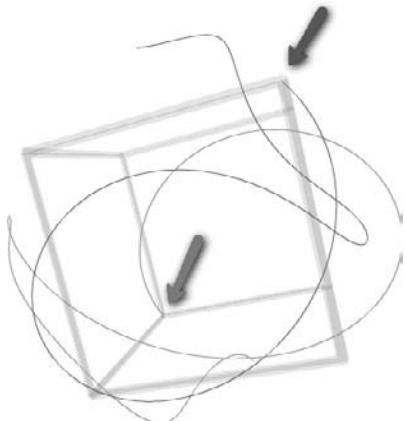
So far, you've used the default projection, which you'll recall has a vanishing point at the center of the stage. You can customize parameters of the projection to get a different feel for the scene. The stage has a perspective projection matrix that it applies to every 3D DisplayObject, projecting the 3D shapes onto the 2D image plane when each frame is rendered. This is like the lens of a camera, which

compresses the scene around it, projecting it onto the film or sensor. And, like a manual lens, you can change some of its properties.

3D points on a cube trace their 2D paths across the screen. Output by Example 15-4.

The projection properties are stored in a `PerspectiveProjection` instance, which is found in the stage's `Transform` object (covered in Chapter 34). Access it with this expression:

```
stage.transform.perspectiveProjection
```

The perspective projection is set on only one display object — the stage — because all 3D objects should be transformed with the same projection — otherwise it's like looking at a scene through mismatched pairs of eyes.

Remember that the viewport is not a camera. It can't be picked up and moved, or rotated. There are two properties that you can change, and one derived property:

- `projectionCenter` — A `Point` representing the vanishing point, in global coordinates.
- `fieldOfView` — The field of view in degrees, as a `Number` between 0 and 180. The field of view controls the impression of depth created by a projection. With a narrow field of view, near 0, depths are flattened, as in a telephoto lens. With a wide field of view, near 180, depths are exaggerated, as in a wide-angle lens.
- `focalLength` — The focal length in pixels, calculated dynamically from the aspect ratio of the stage and the field of view. Set the `fieldOfView` instead.

In Example 15-5, you'll re-create the scene from Example 15-1, but you'll use the mouse and keyboard to control the projection in real time.

EXAMPLE 15-5 http://actionscriptbible.com/ch15/ex5

Changing the Projection

```
package {
  import flash.display.Sprite;
  import flash.events.*;
  import flash.geom.*;
  import flash.text.*;
  import flash.ui.Keyboard;
  public class ch15ex5 extends Sprite {
    protected var tf:TextField;
    public function ch15ex5() {
      var cube:Cube = new Cube(100); addChild(cube);
      cube.x = stage.stageWidth * 0.5; cube.y = stage.stageHeight * 0.5;
      cube = new Cube(100); addChild(cube);
      cube.rotationX = -40; cube.rotationY = 20; cube.rotationZ = 12;
      cube.x = stage.stageWidth * 0.2; cube.y = stage.stageHeight * 0.6;
      cube.z = 250;
      cube = new Cube(100); addChild(cube);
      cube.rotationX = 95; cube.rotationY = -30; cube.rotationZ = 12;
      cube.x = stage.stageWidth * 0.7; cube.y = stage.stageHeight * 0.4;
      cube.z = -250;
      tf = new TextField(); tf.height = 14; tf.x = tf.y = 5;
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.backgroundColor = 0; tf.background = true;
      tf.defaultTextFormat = new TextFormat("_typewriter", 10, 0xffffff);
      addChild(tf);
      stage.addEventListener(MouseEvent.MOUSE_WHEEL, onMouseWheel);
      stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    }
    protected function onMouseMove(event:MouseEvent):void {
      var pp:PerspectiveProjection = root.transform.perspectiveProjection;
      pp.projectionCenter = new Point(event.stageX, event.stageY);
      update(pp);
    }
    protected function onMouseWheel(event:MouseEvent):void {
      adjustFov(event.delta / 2);
    }
    protected function onKeyDown(event:KeyboardEvent):void {
      switch (event.keyCode) {
        case Keyboard.UP: case Keyboard.PAGE_UP: adjustFov(4); break;
        case Keyboard.DOWN: case Keyboard.PAGE_DOWN: adjustFov(-4); break;
      }
    }
    protected function adjustFov(delta:Number): void {
      var pp:PerspectiveProjection = root.transform.perspectiveProjection;
      var fov:Number = pp.fieldOfView + delta;
```

```
      fov = Math.min(Math.max(1, fov), 179);
      pp.fieldOfView = fov;
      update(pp);
    }
    protected function update(pp:PerspectiveProjection):void {
      //set the projection
      stage.transform.perspectiveProjection = pp;

      var ctr:Point = pp.projectionCenter;
      var BR:Point = new Point(stage.stageWidth, stage.stageHeight);
      graphics.clear();
      graphics.lineStyle(0, 0, 0.2);
      graphics.moveTo(0, 0); graphics.lineTo(ctr.x, ctr.y);
      graphics.moveTo(BR.x, 0); graphics.lineTo(ctr.x, ctr.y);
      graphics.moveTo(0, BR.y); graphics.lineTo(ctr.x, ctr.y);
      graphics.moveTo(BR.x, BR.y); graphics.lineTo(ctr.x, ctr.y);
      tf.text = "vanish (" + ctr.x.toFixed() + ", " + ctr.y.toFixed() + ")  " +
        "FOV=" + pp.fieldOfView.toFixed(1) + "°";
    }
  }
}
import flash.display.*;
class Cube extends Sprite {
  public function Cube(size:Number = 50) {
    for (var side:int = 0; side < 6; side++) {
      var face:Sprite = new Sprite();
      face.blendMode = BlendMode.MULTIPLY;
      var brightness:int = 0x80 + 0x80 * Math.random();
      var color:int = brightness << 16 | brightness << 8 | brightness;
      face.graphics.beginFill(color, 0.8);
      face.graphics.drawRect(-size/2, -size/2, size, size);
      face.graphics.endFill();
      addChild(face);
    }
    var S2:Number = size/2;
    getChildAt(0).rotationY = 90; getChildAt(0).x = -S2;
    getChildAt(1).rotationY = -90; getChildAt(1).x = S2;
    getChildAt(2).z = -S2;
    getChildAt(3).rotationY = 180; getChildAt(3).z = S2;
    getChildAt(4).rotationX = 90; getChildAt(4).y = S2;
    getChildAt(5).rotationX = -90; getChildAt(5).y = -S2;
  }
}
```

After playing with this example, you should be able to intuitively grasp how the perspective may be modified. Figure 15-8 shows the field of view near its two extremes, with the same vanishing point. In the wider field of view, it's quite apparent that lines converge toward the vanishing point, along
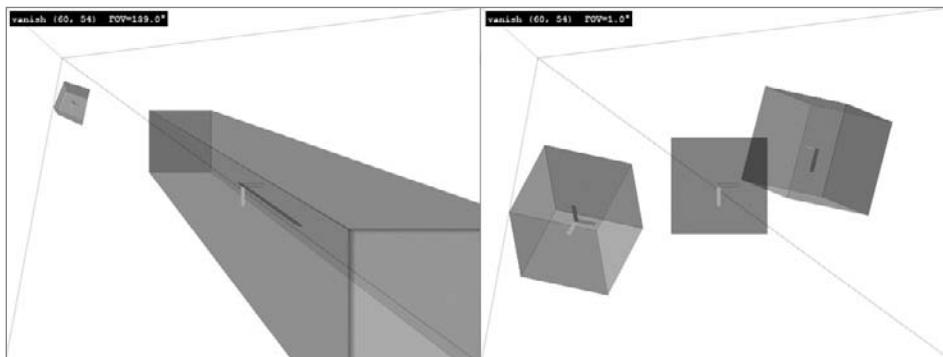
the guides that connect the corners of the viewport with the projection's center. It's also evident that a small difference in z has a big impact with a wide field of view, and almost no impact with a narrow field of view.

FIGURE 15-8

Extreme field of view differences. Output by Example 15-5.

While playing with the example, you may get the impression that moving the vanishing point is analogous to moving and rotating a camera, but it's not so. Moving the vanishing point is odd because it's not something you can do with most fixed cameras (though you could with a tilt-shift lens). That said, you can use it along with a global transformation to be more camera-like.

Software 3D Libraries

There are a few excellent, easy-to-use 3D engines written in pure ActionScript 3.0, all of which can be used with Flash Player 9. Moreover, some of them incorporate the new 3D interfaces in the Flash Player 10 API, offloading some of their work onto Flash Player for greater speed.

Table 15-2 lists the popular 3D engines under active development. The most popular engines (in my subjective opinion) are presented first.

All these engines are incredibly capable and have a ton of impressive features. They have so many features, in fact, that listing them all would be rather dull. Suffice to say, these engines unlock the potential to do amazing, full-on 3D, including meshes, scene import and export, texture mapping, lighting and materials, camera controls, and more. Papervision3D and Away3D, particularly, are under heavy continuing development by a group of very talented coders, and there is a plethora of information on the internet about using both engines.

In addition, there are some special-purpose engines available, such as Five3D for 3D vector graphics, and FFilmation and as3isolib for isometric 3D.

TABLE 15-2

ActionScript 3.0 3D Engines

Engine Name	Site	License	Uses FP10 3D?
Papervision3D	http://blog.papervision3d.org/	MIT License	FP10 branch in development
Away3D	http://away3d.com/	Apache License 2.0	FP10 and FP9 versions
Alternativa3D	http://alternativaplatform.com/en/alternativa3d/	Commercial	FP10 and FP9 versions
Sandy	http://flashsandy.org/	Mozilla Public License	FP9
Yogurt3D	http://yogurt3d.com/	Not Yet Public	

It's worth mentioning that the kinds of 2.5D that Flash Player 10 enables may also be achieved with any of the tools in Table 15-1. You may want to use one of these engines to draw in 2.5D not only for Flash Player 9 compatibility, but for their APIs, which are more geared toward 3D from the get-go. The size of the library and speed may also play into your decision of which tool to use.

Summary

- The display list supports 3D perspective transformations in Flash Player 10 and later.
- More 3D math and transformations are covered in Chapter 34 and Chapter 40.
- The display list still acts like a display list, with layers and nesting and all the same classes.
- Display objects don't have depth; some would call this 2.5D or billboards.
- DisplayObject has properties that let you position, orient, and scale display objects in 3D just like you do in 2D.
- Vector3D represents a point or vector in three dimensions. It has x, y, and z properties.
- Points may be translated from local to global coordinate spaces, from 2D to 3D and back.
- The perspective projection may be modified to change its vanishing point and field of view. Set it on the root display object only.
- Software 3D libraries let you do much more with 3D.

Working with DisplayObjects in Flash Professional

Despite ActionScript 3.0's formidable display list and drawing capabilities, drawing all your application's graphics from scratch using code is far too tedious to be practical. Likewise, loading graphics from many external files can be difficult to manage and may be more trouble than it's worth. In fact, in most production work, developers use Flash Professional's layout, compositing, animation, and drawing tools to prepare assets for use with ActionScript.

In this chapter, you'll learn how Flash Professional works with `DisplayObjects` and how to effectively use embedded and dynamically loaded assets produced in Flash Professional.

The Stage, Symbols, and the Library

Although it may have some extra features, Flash Professional uses the same display list model that you learned about in Chapter 14, "Visual Programming with the Display List." The stage that you see in Flash Professional is the same stage in Flash Player. Anything you can put on the stage in Flash Professional is a `DisplayObject`, and when you run the program, it appears on stage. Simply, Flash Professional acts as a visual editor for the display list.

If you've used Flash Professional, you should be comfortable with *symbols*, which are reusable piece of graphics, sound, or animation. You can repeat the same symbol multiple times. Think of symbols as classes. The symbol's definition in Flash's Library is the class, and symbol instances on stage are instances. Placing a symbol on the stage in Flash has the same effect as instantiating a display object and adding it to the stage at runtime.

The Library in Flash Professional is used to store and organize all the symbols in a given Flash file, whether they are on stage or not. The organization and naming of symbols in the Library has little effect on ActionScript, but as the keeper of all symbols, the Library is of utmost importance.
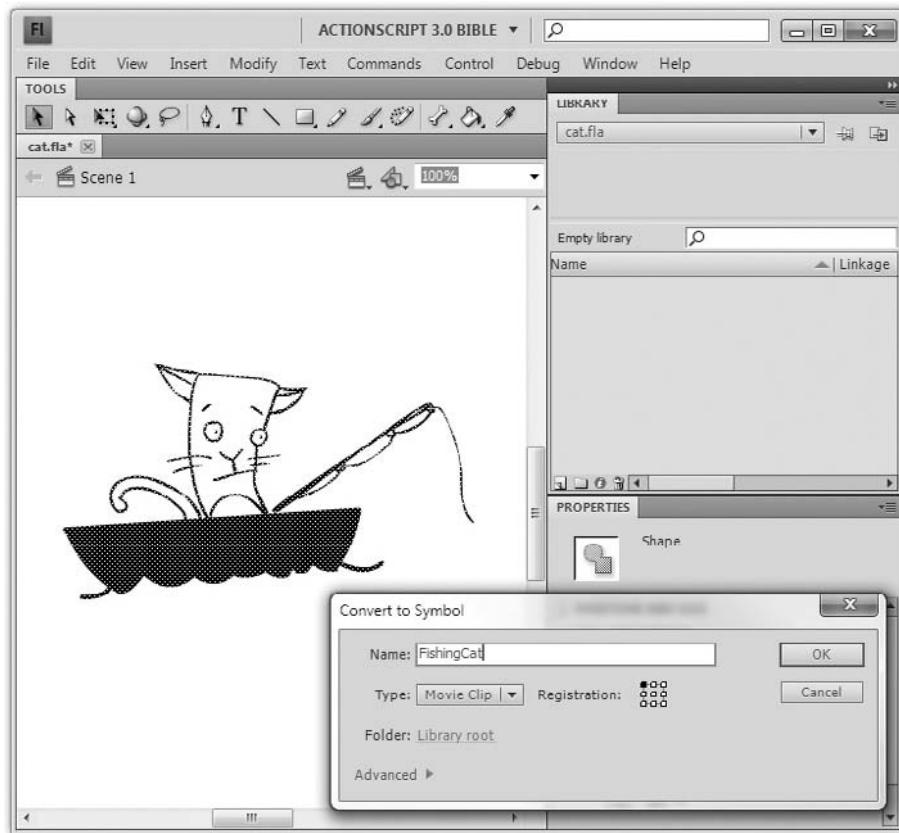
Creating Symbols

Any asset that you create or import using Flash Professional should be made into a symbol. This makes it much easier to play with in ActionScript.

In Figure 16-1, a cat fishing is drawn on the stage. To convert this image into a symbol, first select the drawing and then click Modify ⇨ Convert to Symbol. This opens the Convert to Symbol dialog box. Choose a symbol name (I used `FishingCat`) and the Movie Clip symbol type. It's important to choose this type. The symbol name is unimportant to ActionScript.

FIGURE 16-1

A vector drawing in Flash Professional
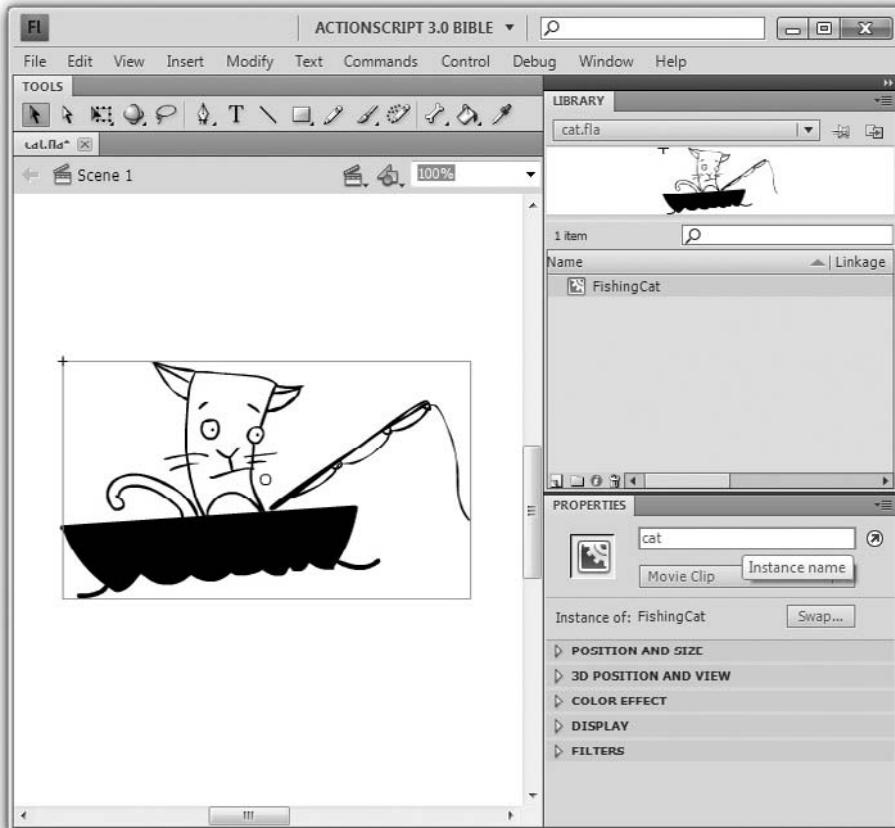
Now that the cat has been converted into a symbol, it should appear in the Library with all symbols. On stage, the Properties panel shows that it is a symbol, specifically a movie clip, and that it is an instance of `FishingCat`.

Named Instances

The first thing you can do to make this symbol accessible to ActionScript is to give it an instance name. Simply type the desired name into the Properties panel, as shown in Figure 16-2.

Naming a symbol instance

All things on stage in Flash Professional are display objects, so you can always access them using display object methods like getChildAt(). But naming an instance makes it much more accessible. The instance name is used to set the name property on the DisplayObject. I've used the instance name cat, so now I can access the cat's display object with getChildByName().

```
trace(getChildByName("cat")); //[object MovieClip]
```

Nested Instances

Perhaps you're making a fishing game with this cat character; you're going to need him to flick the fishing pole back when you press a key. So you go into the cat symbol and convert his fishing rod into its own symbol, naming its instance `fishingRod`. Now you have nested display objects. You'll flick the fishing rod by setting its rotation:

```
getChildByName("cat").getChildByName("fishingRod").rotation = -45;
```

There's a problem with this, however. Not every `DisplayObject` can have children — only those that extend `DisplayObjectContainer`. Even though you know full well that the cat instance has children, `getChildByName()` returns a `DisplayObject`, which doesn't have a `getChildByName()` method. You can solve this problem by casting:

```
var catClip:MovieClip = MovieClip(getChildByName("cat"));
catClip.getChildByName("fishingRod").rotation = -45;
```

This will do the trick, but it gets really tedious when you have to dig down into more complicated nested clips. I like to live life on the wild side and use untyped variables for my display objects sometimes. This stops the compiler from type checking the variable, letting me call any method or access any property it may or may not have.

```
var comp:* = this;
comp.getChildByName("cat").getChildByName("fishingRod").
rotation = -45;
```

Flash Professional can set up an even more convenient way to access instances on the display list. Make sure that Automatically Declare Stage Instances is checked in the ActionScript Settings dialog box, and all named instances will be promoted to properties at compile time. This means that the `FishingCat` symbol will come with a property called `fishingRod`, which is set to the display object named `fishingRod`. Now fishing is as easy as

```
cat.fishingRod.rotation = -45;
```

If you use this option, be aware that these properties can't conflict with existing properties on the associated class. For instance, you shouldn't name an instance `alpha`.

Associating Symbols to Classes
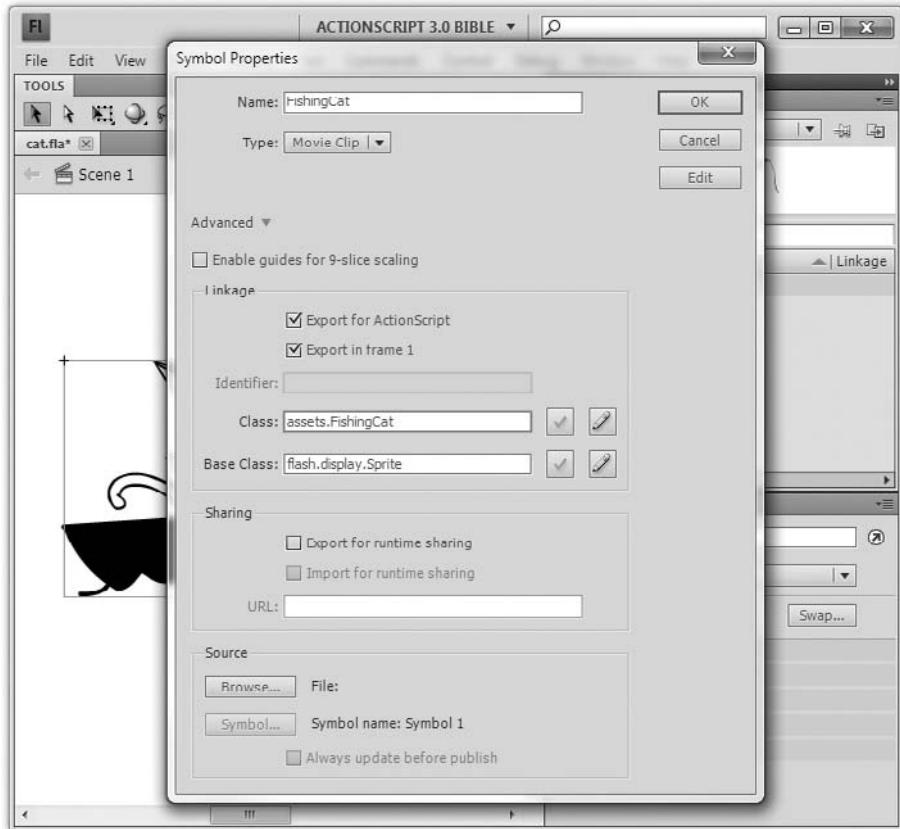
Remember that symbols are like classes. In fact, during compilation, they become subclasses of the appropriate base class for their symbol type. Font symbols become subclasses of `Font`, sound symbols become subclasses of `Sound`, and movie clip symbols become subclasses of `MovieClip`. Associating a symbol with a class is one of the most powerful ways to use it in ActionScript.

You can associate a symbol with a class in the Symbol Properties dialog box. You bring up this dialog box when you're creating the symbol for the first time or at any other time by choosing Properties from the symbol's menu in the Library. You may need to twirl down the Advanced section of the dialog box to see the appropriate items. Check the Export for ActionScript check box, and the Class and Base Class fields will activate, as shown in Figure 16-3.

FIGURE 16-3

Linking the symbol to an ActionScript class

The Class field lets you set the package and name of the class that the symbol will be associated with. I've chosen `assets.FishingCat`. The Base Class field lets you specify which class the symbol will extend. This must be appropriate for the type of the symbol you create. Choose something nonsensical, like `Number`, and Flash Professional will ignore your input. That said, for movie clip symbols, you can use either `MovieClip` or `Sprite` as the base class. Use a `Sprite` if the symbol has only one frame on the timeline.

Now that the symbol is linked to a class, you can create new instances of it just by using its constructor.

```
import assets.FishingCat;
addChild(new FishingCat());
```

Also, any instances of the cat symbol on the stage will be of the `FishingCat` type, although you can still call them by `Sprite` or `DisplayObject`, because `FishingCat` extends both of these.

Writing an Associated Class

If you want, you can write code for that `FishingCat` class. If you compile this class when building the SWF in Flash Professional, the class definition applies to all symbol instances on stage or created at runtime. It's okay to associate the symbol with a class that you don't have code for; the class is generated automatically. The class will be just the same as its base class (`Sprite`) but with the contents of the symbol instead of empty. The same goes for bitmap symbols, sounds, and so on.

If you create a new class called `assets.FishingCat` and store it in `assets/FishingCat.as` relative to the Flash file, Flash Professional uses that definition instead. Don't forget to extend the base class in your definition. Here you make the cat go fishing on a timer.

```
package assets {
  import flash.display.Sprite;
  import flash.utils.setInterval;
  import flash.utils.setTimeout;
  public class FishingCat extends Sprite {
    public function FishingCat() {
      setInterval(rodUp, 1000);
    }
    protected function rodUp():void {
      fishingRod.rotation = -45;
      setTimeout(rodBack, 250);
    }
    protected function rodBack():void {
      fishingRod.rotation = 0;
    }
  }
}
```

Notice that even though I took all that time to define the class, I didn't need to add a definition for the `fishingRod` property. Flash Professional adds this during compilation.

I prefer to keep assets and functionality separate, so I never write code in a symbol class. Instead, I write code that *uses* the symbol, for example, a code-only class that instantiates and manages its own `FishingCat` instance.

Nongraphic Symbol Types

You've seen how to embed vector graphics using Convert to Symbol and Movie Clip symbols. You can use the Library in Flash Professional to embed all kinds of data:

- Font symbols — Extend `flash.text.Font`
- Video symbols — Extend `flash.media.Video`
- Sound symbols — Extend `flash.media.Sound`
- Bitmap symbols — Extend `flash.display.BitmapData`
- Button symbols — Extend `flash.display.SimpleButton`

Use File ⇨ Import to Library in Flash Professional to get these other kinds of data into the Library. For a font, use New Font in the Library menu.

Many of these symbol types provide additional compression options in the symbol's Properties dialog box.

Exporting and Using Assets

Once you have a library full of symbols with linkages, publishing the file generates a SWF or SWC with these symbols. Symbols without linkages aren't included in the export unless they appear on stage or inside a symbol with a linkage.

To export to a SWC instead of a SWF, check the Export SWC option in the Publish Settings dialog box in Flash Professional. Whether you export to a SWF or a SWC impacts how you use the assets in external code.

Using Assets from a SWC

SWCs are linked in at compile time. If you link the SWC statically, its contents (the ones you use, anyway) are included in the end product SWF; link it externally, and it must be loaded along with the end product SWF that depends on it. When you're writing code, an IDE is aware of any classes in linked SWCs, so you can get code completion against these classes.

You can set up Flash Builder to use classes in a SWC by adding the SWC to the ActionScript Build Path of the project's properties. Click the Library path tab and then the Add SWC button.

Use a class defined in a SWC as if it were defined locally.

```
import assets.FishingCat;
addChild(new FishingCat())
```

Using Assets from a SWF

When you use an external asset SWF, your code has no dependencies on the assets, and the asset SWF doesn't have to be available when compiling. You can place the asset SWF anywhere and load it at any time that works for you. This is a great technique for content-heavy web sites; the main SWF can contain the code for the whole site, and you can perform whatever actions you need as you load the required asset SWFs at runtime, like drawing a preloader or going ahead with the parts of the site that can be shown sans assets.

This increased flexibility comes at a cost. To use classes defined in an external SWF, you have to load the SWF and get class references dynamically. You can't use the external classes for typing, because they aren't available at compile time.

You'll learn more about loading files in Chapter 27, "Networking Basics and Flash Player Security." For now, you'll load without explanation. In Example 16-1, you instantiate the `FishingCat` symbol from its asset SWF after loading in said SWF.

Loading a SWF and Instantiating a Symbol

```
package {
  import com.actionscriptbible.Example;
  import flash.display.DisplayObject;
  import flash.display.Loader;
  import flash.events.Event;
  import flash.net.URLRequest;
  import flash.system.ApplicationDomain;
  public class ch16ex1 extends Example {
    protected var loader:Loader;
    public function ch16ex1() {
      trace("Loading assets...");
      loader = new Loader();
      loader.load(new URLRequest(
        "http://actionscriptbible.com/files/ch16assets.swf"));
      loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    }
    protected function onLoad(event:Event):void {
      try {
        var assetsDomain:ApplicationDomain =
          loader.contentLoaderInfo.applicationDomain;
        var FishingCatClass:Class =
          Class(assetsDomain.getDefinition("assets.FishingCat"));
        var cat:DisplayObject = new FishingCatClass();
        cat.y = 100;
        addChild(cat);
        trace("added fishing cat!");
      } catch (error:Error) {
        trace(error);
      }
    }
  }
}
```

Now you can use not just classes defined in the SWF, but the contents of its stage, if you wish. This is as simple as loading the SWF and adding it to your own stage.

Summary

- ActionScript 3.0 stores graphics in class objects. Creating new graphics is as easy as instantiating the class.

- Movie clip symbols bind graphics to a class. This can be a customized class or one that's automatically generated when the SWF is compiled.

Text, Styles, and Fonts

Text is an indispensable aspect of any application, and `TextField` is the base object for dealing with text anywhere in Flash Player. `TextField` handles both labels that display text and interactive text fields that a user can type into, like a form.

To display `TextField` objects, you'll add and remove them from the stage or parent object like any other `DisplayObject`. Because `TextField` extends `InteractiveObject`, it has methods to deal with mouse interaction, and many events it can throw, as well as many of the events I cover in Chapter 21, "Interactivity with the Mouse and Keyboard." Sizing, scaling, moving, and reparenting a `TextField` are all handled in the same way as any other `DisplayObject`.

Unique to `TextField` are methods and properties to control the appearance of text it contains. The formatting can be controlled by `TextFormat` objects, CSS styles, or a subset of HTML. Using the `TextFormat` class, you can set properties for the entire `TextField` or just for certain spans of characters. You'll also look at how to embed fonts and how to control the way those fonts appear using text smoothing properties.

With `TextFields`, you have a lot of control over the display of text in Flash Player. You can get even more control by using the Flash Text Engine and the Text Layout Framework, covered in the next chapter. In this chapter I'll constrain the discussion to classes in the `flash.text` package.

Introducing TextFields

`TextField` is used everywhere you need text displayed. Let's dive right in and start putting text on-screen with `TextFields`.

Creating a New TextField

To create a `TextField`, simply call its constructor, which takes no arguments. The `TextField()` constructor creates a default `TextField` that's 100 × 100 pixels in size. You will almost certainly want to customize the properties of the field later. When you're ready to add the `TextField` to the display list, use `addChild()` to add it to any class that extends `DisplayObjectContainer`.

```
var txt:TextField = new TextField();
addChild(txt);
```

The preceding code snippet assumes that the `TextField` is being created within an object that extends `DisplayObjectContainer` and therefore has a definition for `addChild()`.

Adding and Replacing Text

You add text to a `TextField` by assigning a `String` to its `text` property. After you have some text in the `TextField`, you can add more by appending an additional string to the `text` property, or using the `appendText()` method, as shown in Example 17-1.

EXAMPLE 17-1 http://actionscriptbible.com/ch17/ex1

Appending Text

```
package {
  import flash.display.Sprite;
  import flash.text.TextField;

  public class ch17ex1 extends Sprite {
    public function ch17ex1() {
      var txt:TextField = new TextField();
      txt.text = "Hello World.\n";
      addChild(txt);
      txt.text += "Hello again.\n";
      txt.appendText("Last hello, I promise.\n");
    }
  }
}
```

The `appendText()` method generally runs faster than `+=` and is preferred, as the compiler will remind you.

You can use the `replaceText()` and `replaceSelectedText()` methods to replace text. `replaceText()` needs to be told the range of characters to replace, and `replaceSelectedText()` simply uses the current selection.

Setting a TextField's Size

When you set the height or width of the `TextField`, it displays text only within those boundaries. To change the size of a `TextField`, simply set new values for its `width` and `height` properties, as shown in Example 17-2. Remember that these properties exist on all `DisplayObjects`.

`http://actionscriptbible.com/ch17/ex2`

Resizing Text

```
package {
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.text.TextFieldAutoSize;

  public class ch17ex2 extends Sprite {
    protected const TEXT:String =  "Gaius Marius (157 BCE-January 13, 86 BCE)\
was a Roman general and politician";

    public function ch17ex2() {
      //A – manually sized, small
      var smalltf:TextField = new TextField();
      smalltf.border = true; //we'll turn on the border to see the size
      smalltf.width = 100;
      smalltf.height = 30;
      smalltf.text = TEXT;
      smalltf.y = 0; //appears on top
      addChild(smalltf); //not all text will fit

      //B – manually sized, big
      var bigtf:TextField = new TextField();
      bigtf.border = true;
      bigtf.width = 800;
      bigtf.height = 30;
      bigtf.text = TEXT;
      bigtf.y = 50; //appears in the middle
      addChild(bigtf); //all the text fits, with extra space

      //C – autosizing
      var autotf:TextField = new TextField();
      autotf.border = true;
      autotf.width = autotf.height = 0; //let autosize grow the size from 0
      autotf.autoSize = TextFieldAutoSize.LEFT;
      autotf.text = TEXT;
      autotf.y = 100; //appears on bottom
      addChild(autotf); //all text fits perfectly
    }
  }
}
```

You have another option that you can use to control sizing: `autoSize`. The `autoSize` property of a `TextField` allows it to change size automatically to fit its contents, depending on how the text is aligned and whether the text wraps. You can set `autoSize` to the following values:

- `TextFieldAutoSize.NONE` — The default value. Does not allow the `TextField` to resize automatically.

331

- `TextFieldAutoSize.LEFT` — For use with left-aligned text, will expand the `TextField` toward the right when lines are too long and toward the bottom when there are too many lines to display.
- `TextFieldAutoSize.CENTER` — For use with center-aligned text, will expand the `TextField` equally toward the left and right to make room for lines too long to display and toward the bottom when there are too many lines to display.
- `TextFieldAutoSize.RIGHT` — For use with right-aligned text, will expand the `TextField` toward the left when lines are too long and toward the bottom when there are too many lines to display.

If the `TextField` is set to wrap long lines, the behavior is overridden such that only the bottom is extended when an `autoSize` mode is set. I'll touch on line wrapping shortly.

Sometimes you want your text fields to grow with their contents, like in a tool tip or a caption, but sometimes you want the fields to stay at a fixed size, like in a form field. Using a mixture of `autoSize` and fixed-width `TextFields` lets you do whatever you need.

Setting a TextField's Scaling and Rotation

Both scaling and rotation are behaviors inherited from `DisplayObject`. But both behave in unexpected ways when used with `TextFields` that contain device fonts. You'll learn about device fonts and embedded fonts — and how they behave when rotated — in the section "Fonts." Otherwise, simply use the `rotation`, `scaleX`, and `scaleY` properties inherited from `DisplayObject` that you learned about in Chapter 14, "Visual Programming with the Display List."

Wrapping Text

In Example 17-2, you saw text that spilled over to a second line. With the `TextField` class, you can make sure your text stays all on one line or allow it to use multiple lines, as shown in Example 17-3. Furthermore, you can choose whether those lines should wrap automatically or whether new lines only appear in response to newline characters in the text. These two properties are controlled with the logically named `multiline` and `wordWrap` properties.

EXAMPLE 17-3 `http://actionscriptbible.com/ch17/ex3`

Text on Multiple Lines

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.text.TextField;
  import flash.text.TextFieldAutoSize;
  public class ch17ex3 extends Sprite {
    protected const TEXT:String =  "Ernest Rutherford, (1871- 1937) was a \
New Zealand-born chemist and physicist who fathered nuclear physics.";

    public function ch17ex3() {
      var singlelinetf:TextField = newTF();
      singlelinetf.multiline = false;
```

```
      var multilinetf:TextField = newTF();
      multilinetf.multiline = true;
      multilinetf.wordWrap = false;

      var wraptf:TextField = newTF();
      wraptf.multiline = true;
      wraptf.wordWrap = true;
    }
    protected function newTF():TextField {
      var tf:TextField = new TextField();
      tf.border = true;
      tf.width = 300;
      tf.height = 40;
      tf.text = TEXT;
      tf.y = 60 * numChildren;
      tf.addEventListener(MouseEvent.CLICK, onClick);
      addChild(tf);
      return tf;
    }
    protected function onClick(event:MouseEvent):void {
      var tf:TextField = TextField(event.target);
      tf.autoSize = TextFieldAutoSize.LEFT;
    }
  }
}
```

This example shows that only with `multiline` and `wordWrap` both set to `true` do lines of text wrap automatically. You can click the `TextFields` to turn on auto sizing, which demonstrates that when text wrapping is on, the `TextField` grows in height and wraps instead of simply growing as wide as necessary to accommodate the line.

Caution

Note that if you have newline characters in the text, they display new lines regardless of the `multiline` setting. The `multiline` setting is more useful for input text fields, where it stops the user from adding new lines with the Enter or Return key. ∎

Preventing User Selection

I don't know about you, but one of my pet peeves is when I'm able to select the text of things like menus and preloaders that I would otherwise be normally or habitually clicking without any desire to copy that text — those clicks would be interpreted as selection events and select the menu item or preloader copy instead. On the other hand, another pet peeve of mine is that in so many uses of Flash, I can't copy text that I want to, like the address or phone number of a restaurant (infuriating!) or a quote from a conversation or article.

Well, you can avoid the unbridled wrath of myself and your other users by controlling whether your text can be selected on each `TextField`. Simply use the `selectable` property, which accepts `true` or `false`. Easy!

333

Displaying Multilingual Text and Symbols

Remember from Chapter 6, "Text, Strings, and Characters," that Flash Player is delightfully Unicode-compliant. You can include any Unicode text in your `TextField`s in the same way that you add any text to a `TextField`, as long as it's encoded with UTF-8. If you want to avoid non-ASCII text in your source files or the interface you're loading your text from, you can use an escape sequence with the Unicode code for the non-ASCII characters. You can find a list of all Unicode escape sequences at `http://bit.ly/unicode-chart`.

You can find other references by searching for "Unicode characters list." If you want to use the Greek letter delta, simply type this:

```
txt.text = "The greek letter delta looks like so: \u0395";
```

Caution

Don't forget that Flash Player has to draw the text you request in the font it's set to use, so if you're using a device font, the user's system must have an acceptable system font that includes that glyph, or if you're using an embedded font, that glyph must be embedded. These days, most system fonts include a large subset of Unicode, so using device fonts is a pretty good bet. ■

Text with HTML and CSS

Good news: you can use HTML and CSS in your `TextField`s. Bad news: only limited subsets of HTML and CSS are supported, and those parts that are supported do not always behave correctly. By no means should you feed a `TextField` HTML designed for web browsers and expect it to render correctly! That said, if you use the capabilities provided with their limitations in mind, it can be quite helpful.

When you use HTML in a `TextField`, all you have to do is set the field's `htmlText` property instead of its `text` property. That's it! Interestingly, you can still use the `text` property even when you have HTML displayed inside the `TextField`. This simply returns the text sans tags, which can be useful.

HTML Support in TextField

Following is a list of HTML tags supported by Flash Player. I won't bore you with what these tags do, but if there's something specific about how Flash Player handles them, I'll mention it. If you'd like a refresher on HTML, try w3schools (`http://w3schools.com/`), whose tutorials cut right to the chase.

- `<a>` — Does not automatically underline links, but that can be done with CSS.
- `<b>` — A bold typeface must be available for the font used.
- `<br>` — The text field must be multiline.
- `<font>` — Supports the attributes `face`, `size`, and `color`.
- `<img>` — Embeds JPEG, PNG, and even SWF files. Covered in depth in the next section.
- `<i>` — An italic typeface must be available for the font used.
- `<li>` — Because Flash Player does not recognize ordered and unordered list tags (`<ol>` and `<ul>`), all lists are unordered and drawn with bullets.

- `<p>` — The text field must be multiline. Flash Player can be stubborn with newlines, `<p>` tags, and `<br>` tags, so be careful.

- `<span>` — Only the `class` attribute is supported, not `style`.

- `<textformat>` — Much like `<span>`, applies `TextFormat` formatting instead of CSS formatting to the contained text. Supports a subset of `TextFormat` attributes; others are controlled by other HTML tags like `<font>`. I cover this in greater detail later in the chapter.

- `<u>` — Works as expected.

Although no other tags are supported, unsupported tags will be ignored, so you can still use them if you're trying to repurpose HTML text for both the browser and Flash Player. Furthermore, as in older browsers, XHTML tags will be interpreted like their HTML equivalents, so you can use those too.

Flash Player supports only a subset of named HTML entities. Example 17-4 illustrates all of them.

EXAMPLE 17-4   `http://actionscriptbible.com/ch17/ex4`

HTML Entities

```
var tf:TextField = new TextField();
tf.htmlText = "&lt; &gt; &amp; &quot; &apos;";
trace(tf.text); //< > & " '
```

Finally, note that when you retrieve HTML text from a `TextField`, it's not always the same as the HTML you put in. As Flash Player goes through the `TextField` and figures out how to display all the text, it changes your HTML to its own representation, which you will get back. Mostly you'll notice that your styles are inlined and converted into `TextFormat`s and font tags.

Adding Images or SWF Files to a TextField with `<img>`

The `<img>` tag that you may know from HTML can embed images or SWF files in Flash Player. If you're going to embed SWF files, you can use a few additional properties. Let's walk through the attributes of the `<img>` tag.

- `src` — Specifies the URL to an image or SWF file, or the linkage identifier for a movie clip symbol in the library. This attribute is required; all other attributes are optional. External files (JPEG, GIF, PNG, and SWF files) do not show until they are downloaded completely.

- `width` — The width of the image, SWF file, or movie clip being inserted, in pixels.

- `height` — The height of the image, SWF file, or movie clip being inserted, in pixels.

- `align` — Specifies the horizontal alignment of the embedded image within the text field. Valid values are `left` and `right`. The default value is `left`.

- `hspace` — Specifies the amount of horizontal space that surrounds the image where no text appears. By default this is 8 pixels.

- `vspace` — Specifies the amount of vertical space that surrounds the image where no text appears. By default this is 8 pixels.

- `id` — Specifies the name for the movie clip instance (created by Flash Player) that contains the embedded image file, SWF file, or movie clip. This is useful if you want to control the embedded content with ActionScript.

- `checkPolicyFile` — Controls whether Flash Player will check for a cross-domain policy file on the server associated with the image's domain (as covered in Chapter 27, "Networking Basics and Flash Player Security").

Once you have an image embedded in your `TextField`, if you've set its `id` property, you can retrieve a reference to it using `TextField`'s `getImageReference()` method. This method takes in the ID of an embedded image and, if it can find a matching image, returns it as a `DisplayObject`. If the image is externally loaded, this will be a `Loader` — the contents may or may not be ready by the time you query it, but the `Loader` reference will exist.

Supported CSS Properties

Flash Player only supports a small subset of CSS1. Although you shouldn't expect too much from it, you can get a surprising amount done with just a few simple properties. Using CSS, you can group the text styles used in your application and change them all at once, or even load the styles at runtime so that users of the software can modify the styles without recompiling. I won't go over the meaning of all the CSS properties; numerous resources are available online and offline for these. That said, here are all the properties supported in Flash Player, along with any particulars you need to know:

- `color` — Must be in hexadecimal, such as #ff0000 (red). Does not support shorthand such as #f00 or `red`.

- `display` — Must be `inline`, `block`, or `none`.

- `font-family` — Comma-separated list of font names; these fonts must be available to the currently running code. (See the "Fonts" section later in this chapter for more detail.) The three device fonts have CSS-compatible monikers: `mono` is an alias for the Flash Player device font `_typewriter`, `sans-serif` for `_sans`, and `serif` for `_serif`.

- `font-size` — All font size measurements in Flash Player are in pixels; any unit measurement applied after the numeric value will be ignored and pixels used (so `12px` and `12` are both acceptable).

- `font-style` — Only `normal` and `italic` are supported. Remember that an italic version of the font must be available to use `italic`.

- `font-weight` — Only `normal` and `bold` are supported. Remember that a bold version of the font must be available to use `bold`.

- `kerning` — A property not found in CSS1, set this to `true` to enable font-based kerning. Kerning improves readability of type by nudging certain pairs of letters closer or farther apart. This data is included in the kern table of most nonmonospaced fonts.

- `leading` — Another nonstandard CSS property, it behaves slightly differently from the similar but unsupported `line-height` CSS property. Setting `leading` determines the leading, or space added between each line in multiline text. As with all measurements, units are assumed to be in pixels.

- `letter-spacing` — Units are assumed to be in pixels.

- `margin-left`, `margin-right` — Units are assumed to be in pixels.

- `text-align` — Can be `left`, `center`, `right`, or `justify`.

- `text-decoration` — Only `none` and `underline` are supported. Links are not underlined by default; adding the rule restores this behavior.

  `a {text-decoration: underline}`

- `text-indent` — Units are assumed to be in pixels.

In CSS, you may apply styles to all text that matches the rule. You do this by specifying a *selector* for each rule. In Flash Player, only three CSS selectors are supported.

- Type selectors — These selectors apply to every instance of a particular tag. To set the size of all first-level headers, use `h1 {font-size: 30}`.

- Class selectors — Use these to set the styling for tags that have a particular CSS class set. This lets you apply styling arbitrarily. For example, for the tag:

  `<p class="intro">Once upon a time...</p>`

  all of the following CSS rules would apply:

  ```
  p {} /*applies to all p tags*/
  .intro {} /*applies to all tags with class intro*/
  p.intro {} /*applies to p tags with class intro*/
  ```

- Dynamic pseudo-class selectors — For `<a>` tags, the `:hover` and `:active` pseudo-classes are supported, which let a link have different styles when the mouse is over it and the mouse button is pressing down on it, respectively.

You can also group selectors by separating them with commas. In this way, you can apply a rule to multiple selectors without rewriting the rule. Any more complex selectors, such as those for descendants or children, are not supported.

You can also create styles for tags that are not otherwise recognized by Flash Player. For instance, there is no built-in support for `<h1>` or `<h2>` tags, but you can easily implement them by creating a CSS rule for them that makes them appear the way you want them to. You can even do this for tags that don't exist in HTML, like `<chevre>`, although you're probably better off in general sticking to CSS classes (`<div class="chevre">`) rather than making up tags.

You're probably anxious to see some CSS in action, so let's look at how you can apply CSS to text in ActionScript.

The StyleSheet Object and CSS Parsing

Style sheets in ActionScript 3.0 are implemented by instances of the `flash.text.StyleSheet` class. This class is somewhat of a glorified dictionary: it simply stores CSS rules by their selectors. You can create a new `StyleSheet` with its argument-less constructor, add style rules by calling `setStyle()` and retrieve them with `getStyle()`, or clear out existing styles with `clear()`. Each style rule is stored as an associative array `Object`. So the rule:

```
h1 {color: #333333; font-size: 20; font-family: serif}
```

can be added with the code:

```
var css:StyleSheet = new StyleSheet();
css.addStyle("h1", {color: "#333333", fontSize: 20, fontFamily: "serif"});
```

You'll note that when CSS properties are converted into ActionScript, camelCase takes the place of any hyphens.

Anyway, this is a frustrating way to have to add styles, so thankfully you can simply parse a well-formed CSS file with StyleSheet's parseCSS() method. Just pass this method the contents of the file as a string.

Finally, apply the CSS to a TextField by assigning the StyleSheet object into the TextField instance's styleSheet property. Now you have all the pieces and can put them together into Example 17-5.

EXAMPLE 17-5 http://actionscriptbible.com/ch17/ex5

Style Sheets

```
package {
  import flash.display.Sprite;
  import flash.text.StyleSheet;
  import flash.text.TextField;
  public class ch17ex5 extends Sprite {
    protected const TEXT:String =
"<h1>Quiet Girl</h1>" +
'<h2>by <a href="http://en.wikipedia.org/wiki/Langston_Hughes">' +
"Langston Hughes</a></h2>\n" +
"<p>I would liken you<br>To a night without stars<br>" +
"Were it not for your eyes.<br>I would liken you<br>" +
"To a sleep without dreams<br>Were it not for your songs.";

    protected const CSSRULES:String =
"h1 {font-family: sans-serif; font-size: 14; color: #202020}" +
"h2 {font-family: sans-serif; font-size: 12; color: #303030}" +
"a {font-style: italic; text-decoration: underline}" +
"a:hover {font-style: italic; text-decoration: underline; color: #303080}" +
"p {font-family: serif; text-align: center; font-size: 20; leading: 8}";

    public function ch17ex5() {
      var styleSheet:StyleSheet = new StyleSheet();
      styleSheet.parseCSS(CSSRULES);
      var tf:TextField = new TextField();
      tf.styleSheet = styleSheet;

      tf.width = stage.stageWidth;
      tf.height = stage.stageHeight;
      tf.multiline = true;
      addChild(tf);

      tf.htmlText = TEXT;
    }
  }
}
```

Note that you set the `styleSheet` property individually for each `TextField`. On one hand, this lets you apply whole different style sheets depending on the situation. On the other hand, if you don't need this, and because a `TextField` needs so many lines of initialization, you may also want to abstract your `TextField` generation into its own method or class if you need to create many `TextFields` that have the same styles.

You can also load the CSS string at runtime from an external file, parse it into a `StyleSheet` object, and then apply it to a `TextField`. The great advantage of this is that it allows you to change the appearance of one or multiple `TextFields` without recompiling the SWF file.

Note

Internally, Flash Player converts CSS into `TextFormats`. This is why you won't see your CSS styles when reading out the `htmlText` of a `TextField`. You can create new and exciting style parsing if you want, by extending `StyleSheet` and overriding the `transform()` method, which converts CSS associative arrays into `TextFormats`. Finally, if you want to be really tricky and you prefer CSS syntax to XML syntax, you can hijack `parseCSS()` as a file parser, using it to convert text into structured objects. ∎

Background and Border Treatments

`TextField` objects have `Shape` objects in their background that can be set using the `backgroundColor` and `hasBackground` properties. By default, the background of the `TextField` has an alpha of 0. To change that, you need to set two properties: the `background` Boolean property that, if `true`, means the `TextField` has a background, and the `backgroundColor`, which should be a hexadecimal value in the format of 0xRRGGBB:

```
var txt:TextField = new TextField();
txt.backgroundColor = 0x0000FF; //use a blue background
txt.background = true;
```

You can also create a border for the `TextField` by simply setting `border` and customizing the `borderColor`, which accepts a `uint`, so use hexadecimal colors. Here you'll set the border of the `TextField` to a dark blue.

```
txt.borderColor = 0x000099;
txt.border = true;
```

Of course, you can always create more complex borders and backgrounds for your `TextFields` by simply placing them underneath the `TextField` using normal display list operations.

Styling Text with TextFormats

The `TextFormat` object is another way to give text in your `TextField` a specific look. If you're using HTML, you may find it easier to use the `StyleSheet` object. If you're formatting arbitrary ranges of text — for example, a line, or the first 100 characters — the `TextFormat` object might make more sense to use. `TextFormat` is an object whose properties control formatting of the text it is applied to.

To use `TextFormat`, create a new `TextFormat` object and set any of its instance properties you'd like to apply to the text. To apply the `TextFormat` object to the `TextField`, either set it to the

TextField's `defaultTextFormat` property to apply the styles to any new text added, as shown in Example 17-6.

Collected Snippets on Applying TextFormats

```
var tf:TextField = new TextField();
tf.autoSize = TextFieldAutoSize.LEFT;
addChild(tf);

var fmt:TextFormat = new TextFormat();
fmt.font = "_sans";
fmt.bold = true;
fmt.size = 24;
tf.defaultTextFormat = fmt;
//note that you have to set the default format BEFORE setting the text
//it applies to all NEW text added.
tf.text = "be bold.";
```

You can also apply the `TextFormat` to specific characters in the `TextField`, using `setText Format()` and passing in integers to mark the first and last characters that you want to receive the formatting. If you don't specify the last index, the format is applied to all succeeding characters.

```
tf = new TextField();
tf.autoSize = TextFieldAutoSize.LEFT;
tf.y = 50;
tf.text = "RED GREEN BLUE";
addChild(tf);

var redFmt:TextFormat = new TextFormat("_sans", 20, 0xff0000);
var greenFmt:TextFormat = new TextFormat("_sans", 20, 0x00ff00);
var blueFmt:TextFormat = new TextFormat("_sans", 20, 0x0000ff);
var t:String = tf.text;
tf.setTextFormat(redFmt, t.indexOf("RED"), t.indexOf("RED")+3);
tf.setTextFormat(greenFmt, t.indexOf("GREEN"), t.indexOf("GREEN")+5);
tf.setTextFormat(blueFmt, t.indexOf("BLUE"), t.indexOf("BLUE")+4);
```

One thing to notice in this example is that you can set all of a `TextFormat`'s properties right from the constructor, which can save you lines of code if, say, you don't want an ActionScript book you're writing to weigh more than a compact car.

Additionally, `setTextFormat()`, when only sent a `TextFormat` and no start or end parameters, applies that format to the entire `TextField`. You can see this in action in Chapter 9, "Vectors," Example 9-3 (http://actionscriptbible.com/ch9/ex3).

The `TextFormat` class has a slew of properties that you can use to apply various formatting to text. The following sections detail each of these properties.

align

You can use the `align` property to place the text relative to the right and left edges of the `TextField` object's bounding box. The property can have one of the values `left`, `right`, `center`, or `justify`. These properties only make sense when applied to a range of text that spans at least one full line, preferably a whole paragraph.

blockIndent

The `blockIndent` property has an effect on text only when the text is aligned left. In that case, the `blockIndent` property indents the entire block of text inward relative to the left margin. The value should be a number indicating how many pixels you want to indent the text by. See also the `leftMargin` and `rightMargin` properties.

bold

The `bold` property applies bold formatting to the targeted text. To turn bold formatting on, use a Boolean value of `true`. To turn bold off, use a Boolean value of `false`. By default, this property is defined with a `null` value, which produces the same effect as `false`.

bullet

The `bullet` property adds a bullet character (●) in front of the text if the property's value is set to `true`. You can turn off bullet formatting by assigning a `false` value to the property. By default, this property has a value of `null`. The font face used for the bullet character is the same as that defined for other text in the `TextFormat` object (via the `font` property, discussed in this section under "font"). The bullet points are placed a super-arbitrary 19 pixels from the left margin of the field, affected only by the left margin settings. (Properties such as `blockIndent` don't have an effect when bullet points are used.) The bulleted text is spaced 15 pixels to the right of the bullet point.

The following code displays a list of bulleted text:

```
var tf:TextField = new TextField();
tf.multiline = true;
tf.border = true;
tf.wordWrap = true;
tf.text = "a\nb\nc\nd";
var fmt:TextFormat = new TextFormat();
fmt.bullet = true;
tf.setTextFormat(fmt);
addChild(tf);
```

color

This property controls the font color of the targeted text. The value for this property should be hexadecimal RGB. (Note that you can set the font color throughout the entire `TextField` by using the `TextField` property `textColor`.) The following code displays red text:

```
var tf:TextField = new TextField();
tf.text = "red text";
var format:TextFormat = new TextFormat();
```

341

```
format.color = 0xFF0000;
tf.setTextFormat(format);
addChild(tf);
```

font

The `font` property controls the font face used for the text. This property uses a string value, indicating the name of the font. The name that you use can depend on how you are working with the font in Flash. By default, the `font` property has a value of `null`, which results in the default font being used. The font face can be applied only if the user has the font installed on his system or if the font has been embedded or shared with the Flash application.

I discuss fonts in depth in the section "Fonts."

indent

The `indent` property controls the spacing applied from the left margin to the first line of text within a paragraph. A *paragraph* is defined as any text that precedes a newline. This property uses pixel units. The default value is `null`.

italic

The `italic` property controls whether the targeted text is displayed in italic. If the property is set to `true`, the text appears italic. If the property is set to `false`, the text appears normal. By default, this property has a value of `null`, which achieves the same effect as a value of `false`. Keep in mind that an italic version of the font must be available.

leading

The `leading` property controls the spacing inserted between each line of text. The values for this property are pixel-based. By default, the value of this property is `null`. You can compress lines as well by using negative leading.

letterSpacing

The `letterSpacing` property controls the tracking, or amount of space between every character in a run of text. A value of `0`, or the default, displays the text in the desired font as you would see it in any program or on the page. Tweaking this value artificially condenses (when negative) and extends (when positive) text. This is usually used to fit more or less body copy on a line, but it can be used stylistically as well. Fractional numbers may be used for this property.

leftMargin

The `leftMargin` property determines the spacing (in pixels) inserted between the text and the left border of the `TextField` object. By default, the value of this property is `null`, which achieves the same effect as a value of `0`.

The `blockIndent` and `leftMargin` properties are added together to determine a given line's indentation.

rightMargin

The `rightMargin` property controls the spacing (in pixels) inserted between the text and the right border of the `TextField` object. By default, the value of this property is `null`, adding no space.

size

The `size` property determines the font size (in pixels) of the text.

tabStops

The `tabStops` property defines a custom array specifying the values used by tabs within the text. The first element of the array specifies the spacing (in pixels) to use for the first tab character in succession. The second element specifies the spacing to use for the second tab character in succession, and so on. The value of the last element in the array is used for all subsequent tab characters. The default value for `tabStops` is null. When the property has a value of `null`, the default value of four pixels is used between each successive tab character. However, using the `tabStops` property, you can specify how ordered tabs are spaced within text.

For example, you can create a `TextFormat` object that uses a tab spacing of 10 pixels for the first tab, a tab spacing of 50 pixels for the second tab (in succession), and a tab spacing of 150 pixels for the third tab. The following code does just that:

```
var tf:TextField = new TextField();
tf.width = 200;
tf.multiline = true;
tf.border = true;
tf.text = "\ta\n";
tf.text += "\t\tb\n";
tf.text += "\t\t\tc";
var format:TextFormat = new TextFormat();
format.tabStops = [10, 50, 150];
format.align = "left";
tf.setTextFormat(format);
addChild(tf);
```

target

The `target` property works in conjunction with the `url` property. You can specify a string value for the `target` property that indicates the name of the browser window (or frame) where the URL specified in the `url` property should appear. You can use the predefined target values of `"_blank"` (new empty browser window), `"_self"` (the current frame or window), `"_parent"` (the parent frame or window), or `"_top"` (the outermost frame or window), or you can use a custom browser window or frame name (as assigned in the HTML document or JavaScript). If you use the `url` property without specifying a value for the `target` property, the URL loads into the current frame or window (`"_self"`).

underline

The `underline` property can add an underline to text. When this property is set to `true`, an underline appears with the text. When it is set to `false`, any underlines are removed. By default, the value of this property is `null`, which has the same effect as a value of `false`.

url

The `url` property allows you to add a hyperlink to text. Flash Player does not provide an immediate indication that the `url` property is in use for a given range of text — you may want to change the color and add an underline to the affected text to make the link more apparent to the user. However, the mouse pointer automatically changes to the hand icon when the mouse rolls over the linked text.

The following code applies a hyperlink to a portion of the text:

```
var tf:TextField = new TextField();
tf.multiline = true;
tf.border = true;
tf.wordWrap = true;
tf.text = "Visit the Web site";
var format:TextFormat = new TextFormat();
format.url = "http://actionscriptbible.com/";
format.target = "_blank";
format.underline = true;
tf.setTextFormat(format, 10, 18);
addChild(tf);
```

Input TextFields

When you want to accept text input from a user, there's just one way to go, and that's an input `TextField`. Input `TextFields` are also `TextField` instances. They can do all the things you've seen, but the end user can only type into an input `TextField`.

The Three Kinds of Text Fields

Flash Player has three basic kinds of text fields. In the next chapter, you'll see that there are other text layout and display facilities when using the Flash Text Engine. For now, let's quickly review the three basic types.

The kind of text field you've seen so far has been a dynamic `TextField`. Dynamic `TextFields` can be controlled by ActionScript code. You can read and set their contents, display them with embedded or device fonts, and manipulate and interact with them in all the ways seen here.

Input `TextFields` are used for user input. They are also instances of `flash.text.TextField`. They add the ability to accept keyboard input.

The third kind I haven't discussed here are static text fields. These are text fields that ActionScript does not control. They can only be created by SWF authoring tools like Flash Professional. If your SWF is built with Flash or you've loaded an external SWF, using ActionScript, you can find these static text fields on the display list. They are instances of `flash.text.StaticText`, not `TextField`, so none of the properties and methods I've discussed here apply to them. Because they extend `DisplayObject`, you can move, scale, rotate, and reparent them; furthermore, you can get their text with the read-only `text` property, but that's all.

You differentiate between the two kinds of `TextField` instances using the `type` property.

Making a TextField an Input Field

To change a new or existing dynamic text field to an input text field, simply set its `type` property to the constant `TextFieldType.INPUT`. Once you've done that, you've got an input `TextField`. Any text that you specify will appear at the beginning of the field and will be editable by the user. To set the type to a dynamic text field, use the constant `TextFieldType.DYNAMIC.` This is the default for all new `TextField` instances.

```
var tf:TextField = new TextField();
tf.type = TextFieldType.INPUT;
tf.text = "Enter your name here";
tf.border = true;
addChild(tf);
```

As soon as the user begins typing in your input `TextField`, he adds onto the original text. As you learn more capabilities of input text fields, we'll build up a more complicated example that is just what you'd use in a form.

Restricting User Input

What if you want to control the characters that a user can put into a `TextField`? If you were to have a `TextField` in which your users enter their phone numbers, you would want to prevent them from entering letters. You can do this using the `restrict` property of the `TextField`. When you set the restrict characters property, it restricts keyboard input only; ActionScript code may still put any text into the text field.

The `restrict` property is set to a string that enumerates the characters that may be typed into the `TextField`. For example:

```
var textField:TextField = new TextField();
textField.type = TextFieldType.INPUT;
textField.restrict = "abc";
addChild(textField);
```

This code only allows the letters *a*, *b*, or *c* to be typed into the field, although the user can type them in any combination, frequency, or order. Allow a range of characters by using a hyphen:

```
textField.restrict = "0-9";
```

The preceding code allows all numbers and nothing else. The next snippet of code allows all numbers and uppercase letters from *A* to *F*, which would be handy if you were trying to keep your text field to valid hexadecimal colors.

```
textField.restrict = "0-9A-F";
```

If you'd rather specify the illegal characters than the legal ones, use the caret (^), which marks anything succeeding it as disallowed and anything preceding it as acceptable:

```
textField.restrict = "^4";
```

This allows any character except the number 4, which can be bad luck. So far so good, but what if you want to allow the hyphen because you want users to be able to enter their phone number? You can't just throw a hyphen in there because it has a special meaning. You can escape it by prefacing it with two backslashes (\\). You need two because you need to use the actual backslash character, which itself needs to be escaped in the `String`. Awful, I know.

```
textField.restrict = "0-9\\-";
```

Now users can enter numbers and dashes and nothing else.

The other control that you have over the user's input into the input `TextField` is the `maxChars` property. This allows you to set the maximum number of characters that a `TextField` can contain. You may recall the phone number example from earlier in the chapter. In that instance, if you were accepting U.S. numbers of the format `xxx-xxx-xxxx`, you would limit your `TextField` to 12 characters, so you would use the following:

```
textField.maxChars = 12;
```

Let's put all this together into Example 17-7. You'll create a form and then later use `TextField` events to make it more user-friendly.

EXAMPLE 17-7   `http://actionscriptbible.com/ch17/ex7`

User Input

```
package {
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.text.TextFieldType;
  import flash.text.TextFormat;

  public class ch17ex7 extends Sprite {
    protected const INFMT:TextFormat = new TextFormat("_sans", 12, 0x505050);
    protected var nameTF:TextField;
    protected var phoneTF:TextField;
    protected var emailTF:TextField;

    public function ch17ex7() {
      nameTF = makeInputTextField();
      nameTF.text = "Your name";
      nameTF.maxChars = 40;

      phoneTF = makeInputTextField();
      phoneTF.text = "Your phone number";
      phoneTF.maxChars = 20;
      phoneTF.restrict = "0-9()\\-";

      emailTF = makeInputTextField();
      emailTF.text = "Your email address";
      emailTF.maxChars = 40;
      emailTF.restrict = "a-zA-Z0-9_\\-+=~!@#$%\\^.";
    }
```

```
  protected function makeInputTextField():TextField {
    var tf:TextField = new TextField();
    tf.type = TextFieldType.INPUT;
    tf.border = true;
    tf.defaultTextFormat = INFMT;
    tf.width = 300;
    tf.height = 18;
    tf.y = numChildren * 26;
    addChild(tf);
    return tf;
  }
 }
}
```

Tab-Accessible Input Text Fields

One of the things to keep in mind when designing a user interface, no matter how simple, is what the user is accustomed to doing. Many people are used to being able to tab through a group of TextFields to change focus from one TextField to another. If you're on the TextField at the top of the page, you tab to get the next lowest. If you're in the TextField at the bottom of the page, you tab to go to the Submit button. Because some users are accustomed to it, they might be pleasantly surprised to find that they can tab through the TextField object, although more likely than not they won't notice. Because they assume that your app is tab-enabled, they'll notice only if it isn't. Such is the lot of a UI designer.

Input text fields should be tab-enabled by default. As you can see in the previous example, the tab key works fine to tab through the fields. In Chapter 21, you'll see how to enable tabbing to an InteractiveObject by setting its tabEnabled property to true. Then, if necessary, you can manually change the tab ordering by assigning ordered integers to the InteractiveObject's tabIndex properties. It's much preferable to add InteractiveObjects to the display list in the correct order and use the automatic tab ordering than to have to design your own, but it's sometimes necessary.

Password Text Fields

You can create password input fields that obscure the characters being typed simply by setting the TextField's displayAsPassword parameter to true.

Interaction with TextField Events

In addition to events inherited from InteractiveObject and DisplayObject, TextField dispatches four kinds of event. With these events, you can make your TextFields much more reactive.

focusIn and focusOut Events

The FocusEvent.FOCUS_IN and FocusEvent.FOCUS_OUT events are inherited from InteractiveObject but are particularly relevant to input text fields. These events are broadcast any time that the TextField receives or loses focus from the user, respectively. You can use this to

update the form you started building in Example 17-7, taking the user-friendly prompts and clearing them out when the user focuses the field, as shown in Example 17-8.

EXAMPLE 17-8 http://actionscriptbible.com/ch17/ex8

User Input with Focus

```
package {
  import flash.display.Sprite;
  import flash.events.FocusEvent;
  import flash.text.TextField;
  import flash.text.TextFieldType;
  import flash.text.TextFormat;
  import flash.utils.Dictionary;

  public class ch17ex8 extends Sprite {
    protected const INFMT:TextFormat = new TextFormat("_sans", 12, 0, true);
    protected const OUTFMT:TextFormat =
      new TextFormat("_sans", 12, 0x808080, false);
    protected var prompts:Dictionary;
    protected var nameTF:TextField;
    protected var phoneTF:TextField;
    protected var emailTF:TextField;

    public function ch17ex8() {
      prompts = new Dictionary();

      nameTF = makeInputTextField();
      nameTF.text = "Your name";
      nameTF.setTextFormat(OUTFMT);
      nameTF.maxChars = 40;
      prompts[nameTF] = nameTF.text;

      phoneTF = makeInputTextField();
      phoneTF.text = "Your phone number";
      phoneTF.setTextFormat(OUTFMT);
      phoneTF.maxChars = 20;
      phoneTF.restrict = "0-9()\\-";
      prompts[phoneTF] = phoneTF.text;

      emailTF = makeInputTextField();
      emailTF.text = "Your email address";
      emailTF.setTextFormat(OUTFMT);
      emailTF.maxChars = 40;
      emailTF.restrict = "a-zA-Z0-9_\\-+=~!@#$%\\^.";
      prompts[emailTF] = emailTF.text;
    }

    protected function onFocusIn(event:FocusEvent):void {
      var tf:TextField = TextField(event.target);
      if (tf.text == prompts[tf]) {
        tf.text = "";
```

```
    }
    tf.setTextFormat(INFMT);
    tf.defaultTextFormat = INFMT;
  }

  protected function onFocusOut(event:FocusEvent):void {
    var tf:TextField = TextField(event.target);
    if (tf.text == "") {
      tf.text = prompts[tf];
      tf.setTextFormat(OUTFMT);
    }
  }

  protected function makeInputTextField():TextField {
    var tf:TextField = new TextField();
    tf.type = TextFieldType.INPUT;
    tf.border = true;
    tf.width = 300;
    tf.height = 18;
    tf.y = numChildren * 26;
    tf.addEventListener(FocusEvent.FOCUS_IN, onFocusIn);
    tf.addEventListener(FocusEvent.FOCUS_OUT, onFocusOut);
    addChild(tf);
    return tf;
  }
  }
}
```

The form fields take away the prompts as soon as you tab or click into them; likewise, they restore the prompts if you leave the field empty. Furthermore, the example uses a different `TextFormat` for the prompts to better differentiate prompt text and input text.

Text Input Events

Two different kinds of events are fired when the user modifies the contents of an input `TextField`, `Event.CHANGE` and `TextEvent.TEXT_INPUT`. There are three main differences between these events.

First, the `TextEvent.TEXT_INPUT` event is fired when the user types text in the field, whether that changes the contents of the field or not. For example, typing illegal characters inside a restricted input field doesn't change the text because these characters are ignored. However, the `TEXT_INPUT` event is fired. This also means that when the user *deletes* text with the backspace key or the clipboard, the `TEXT_INPUT` event is not fired. The `Event.CHANGE` event, on the other hand, is dispatched only when the actual contents have changed, and it doesn't care if that's from typing, deleting, cutting, or pasting.

Second, the `TEXT_INPUT` event is fired before the contents of the field change, whereas the `CHANGE` event is fired after the contents have changed. So when you look at the `text` property of a `TextField` while it dispatches a `TEXT_INPUT` event, it contains the text in that field immediately prior to the event occurring. During a `CHANGE` event, the `text` property is already up-to-date.

349

Finally, the `TEXT_INPUT` event tells you what text is about to be added. You can find this in the `text` property of the `TextEvent`. Because it happens before the text is actually added, and it tells you what text is about to be added, you can cancel the event with `preventDefault()` to prevent the user's text addition from taking place. In other words, by listening to the `TEXT_INPUT` event, you can gain much finer or context-sensitive control over what goes into a `TextField` when a user types into it. This could be used to automatically add dashes to license keys or automatically capitalize letters as you type them.

If you just want to know when the contents of the `TextField` have changed, however, `Event.CHANGE` is a much simpler event to listen to. You can use this to do on-the-fly validation of form fields. Example 17-9 adds a Submit button to the form you built in Examples 17-7 and 17-8 and only lets the user click this button when all the fields are correct.

EXAMPLE 17-9   `http://actionscriptbible.com/ch17/ex9`

Change Events

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.FocusEvent;
  import flash.text.*;
  import flash.utils.Dictionary;

  public class ch17ex9 extends Sprite {
    protected const INFMT:TextFormat = new TextFormat("_sans", 12, 0, true);
    protected const OUTFMT:TextFormat = new TextFormat(
      "_sans", 12, 0x808080, false);
    protected var prompts:Dictionary;
    protected var nameTF:TextField;
    protected var phoneTF:TextField;
    protected var emailTF:TextField;
    protected var okButton:Sprite;

    public function ch17ex9() {
      prompts = new Dictionary();

      nameTF = makeInputTextField();
      nameTF.text = "Your name";
      nameTF.setTextFormat(OUTFMT);
      nameTF.maxChars = 40;
      prompts[nameTF] = nameTF.text;

      phoneTF = makeInputTextField();
      phoneTF.text = "Your phone number";
      phoneTF.setTextFormat(OUTFMT);
      phoneTF.maxChars = 12;
      phoneTF.restrict = "0-9\\-";
      prompts[phoneTF] = phoneTF.text;

      emailTF = makeInputTextField();
      emailTF.text = "Your email address";
```

```
    emailTF.setTextFormat(OUTFMT);
    emailTF.maxChars = 40;
    emailTF.restrict = "a-zA-Z0-9_\\-+=~!@#$%\\^.";
    prompts[emailTF] = emailTF.text;

    okButton = makeOkButton();
    validate();
}
protected function onFocusIn(event:FocusEvent):void {
    var tf:TextField = TextField(event.target);
    if (tf.text == prompts[tf]) {
        tf.text = "";
    }
    tf.setTextFormat(INFMT);
    tf.defaultTextFormat = INFMT;
}
protected function onFocusOut(event:FocusEvent):void {
    var tf:TextField = TextField(event.target);
    if (tf.text == "") {
        tf.text = prompts[tf];
        tf.setTextFormat(OUTFMT);
    }
}
protected function onChange(event:Event):void {
    validate();
}
protected function validate():void {
    if (isValid) {
        okButton.alpha = 1;
        okButton.mouseEnabled = true;
    } else {
        okButton.alpha = 0.2;
        okButton.mouseEnabled = false;
    }
}
protected function get isValid():Boolean {
    if (isEmpty(nameTF)) return false;
    if (isEmpty(phoneTF)
        || phoneTF.text.match(/^\d{3}-\d{3}-\d{4}$/) == null) return false;
    if (isEmpty(emailTF)
        || emailTF.text.match(/^\w+@\w+\.[a-z]{2,6}$/) == null) return false;
    //warning, this regex is not industry grade!
    return true;
}
protected function isEmpty(tf:TextField):Boolean {
    return (tf.text.replace(/\s*/, "").length == 0
        || tf.text == prompts[tf]);
}
protected function makeInputTextField():TextField {
    var tf:TextField = new TextField();
    tf.type = TextFieldType.INPUT;
    tf.border = true;
    tf.width = 300;
```

continued

351

EXAMPLE 17-9 *(continued)*

```
      tf.height = 18;
      tf.y = numChildren * 26;
      tf.addEventListener(FocusEvent.FOCUS_IN, onFocusIn);
      tf.addEventListener(FocusEvent.FOCUS_OUT, onFocusOut);
      tf.addEventListener(Event.CHANGE, onChange);
      addChild(tf);
      return tf;
    }
    protected function makeOkButton():Sprite {
      var w:Number = 100, h:Number = 20;
      var btn:Sprite = new Sprite();
      var label:TextField = new TextField();
      label.width = w;
      label.height = h;
      label.defaultTextFormat = new TextFormat("_typewriter", 14, 0, true,
        null, null, null, null, TextFormatAlign.CENTER);
      label.text = "Submit";
      btn.buttonMode = true;
      btn.mouseChildren = false;
      btn.addChild(label);
      btn.graphics.lineStyle(1, 0x202030);
      btn.graphics.beginFill(0xb0b0b0, 1);
      btn.graphics.drawRect(0, 0, w, h);
      btn.graphics.endFill();
      btn.y = this.height + 20;
      addChild(btn);
      return btn;
    }
  }
}
```

This is the most complete incarnation of the form yet, with real-time validation. You used the change event to revalidate every time the `TextFields` changed. You also made extensive use of regular expressions, which you learned about in Chapter 12, "Regular Expressions."

Link Events

By listening for the `TextEvent.LINK` event, you can be notified when the user clicks a link in a `TextField`. If you design the targets of the links in a particular way, you can get data out of each link rather than having them navigate a web browser to a URL. This means you can trigger ActionScript events not just from clicks on `TextFields` but from clicks on specific words in a `TextField`. For example, you could write a simple menu using `TextField` to lay out all the labels as hyperlinked text, rather than manually adding multiple `TextFields` with their own individual event listeners.

To route a click on an `<a>` tag through the event flow, set the link to any URI with the `event:` scheme. This is a virtual scheme used to notify Flash Player that the link should dispatch an event.

Any parts of the URI after event: are stored in the event object's text property, so you can determine for yourself how to communicate through links. For example, you could decide to send a method name and arguments in the link by simply separating them by commas, as shown in Example 17-10.

EXAMPLE 17-10   `http://actionscriptbible.com/ch17/ex10`

Using Link Events

```
package {
  import flash.display.Sprite;
  import flash.events.TextEvent;
  import flash.text.*;
  public class ch17ex10 extends Sprite {
    protected var tf:TextField;
    public function ch17ex10() {
      tf = new TextField();
      tf.selectable = false;
      tf.defaultTextFormat = new TextFormat("_sans", 14);
      tf.multiline = tf.wordWrap = true;
      tf.width = 200;
      tf.htmlText = "This text controls itself. You could turn it" +
' <a href="event:color,0xff0000"><u>red</u></a> or' +
' <a href="event:color,0x00ff00"><u>green</u></a>. Or you could' +
' <a href="event:move,20,0"><u>move it right</u></a> or' +
' <a href="event:move,0,20"><u>move it down</u></a>.';
      tf.addEventListener(TextEvent.LINK, onLink);
      addChild(tf);
    }
    protected function onLink(event:TextEvent):void {
      var args:Array = event.text.split(",");
      if (args.length < 1) return;
      switch (args.shift()) {
        case "color": colorText.apply(this, args); break;
        case "move": moveText.apply(this, args); break;
      }
    }
    protected function colorText(colorStr:String):void {
      tf.setTextFormat(new TextFormat(null, null, parseInt(colorStr, 16)));
    }
    protected function moveText(xStr:String, yStr:String):void {
      tf.x += parseFloat(xStr);
      tf.y += parseFloat(yStr);
    }
  }
}
```

You could use the LINK event to do even crazier things, like store serialized objects (`http://bit.ly/store-objects-in-links`).

Scroll Events

The `Event.SCROLL` event is dispatched after a `TextField` scrolls. I'll cover this along with more about `TextField` scrolling in the later section "Scrolling text." If you've created a scrollbar, and the user used the scroll wheel to scroll the `TextField`, the scrollbar needs to be notified so that it can update to reflect the new scroll position. You would accomplish that using this event, which you'll see in Example 17-13.

Interactive Typography

Believe it or not, `TextField` still has plenty of tricks left up its sleeve. In this section I'll cover functionality you need to map back and forth between the graphical domain, dominated by pixels and `DisplayObjects`, and the text domain, made only of letters. By doing so, you can create graphics that are aware of text and text that is aware of graphics.

Text by Lines and Paragraphs

When using `TextField`'s text wrapping facilities, Flash Player decides what word should go where. After all is said and done, you can easily see on your screen where a particular word went on the screen, but how can you tell in code?

Flash Player lets you query multiline `TextFields` for a variety of information on the lines of text they contain. The available methods and properties include these:

- `numLines` — The total lines of text.
- `getLineIndexOfChar(charIndex:int):int` — Given a character position in the source text, determines which line that character ended up on. If you want to find out what line a word is on, you can use this function with the character index of its first letter.
- `getLineLength(lineIndex:int):int` — Returns how many characters ended up fitting on a specific line.
- `getLineOffset(lineIndex:int):int` — Given a line number, returns the index in the source text of the character that begins that line. In other words, from where in the text does this line start?
- `getLineText(lineIndex:int):String` — Given a line number, returns what text ended up on that line.
- `getFirstCharInParagraph(charIndex:int):int` — Given a position in the source text, returns the position of the paragraph that contains that character. In other words, rewinds backward from any character to the beginning of its paragraph.
- `getParagraphLength(charIndex:int):int` — Given a position in the source text, tallies up the characters in the containing paragraph.

These methods are most useful when you're using plaintext; HTML contains characters that are not rendered, so character indices in HTML text are misleading.

In Example 17-11, suppose you want to set the characters in the line underneath the user's mouse to red. You'll use several text-by-lines methods to determine where to apply the formatting.

http://actionscriptbible.com/ch17/ex11

Line Formatting

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.text.*;
  public class ch17ex11 extends Sprite {
    protected var tf:TextField;
    protected const RED:TextFormat = new TextFormat("_serif", 28, 0xff0000);
    protected const BLACK:TextFormat = new TextFormat("_serif", 28, 0);

    public function ch17ex11()
    {
      tf = new TextField();
      tf.multiline = true;
      tf.wordWrap = true;
      tf.width = 450;
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
      tf.addEventListener(MouseEvent.MOUSE_OUT, onMouseOut);
      tf.defaultTextFormat = BLACK;
      tf.text = "Alice was beginning to get very tired of sitting by her \
sister on the bank, and of having nothing to do: once or twice she had peeped \
into the book her sister was reading, but it had no pictures or conversations \
in it, 'and what is the use of a book,' thought Alice, 'without pictures or \
conversation?'";

      addChild(tf);
    }
    protected function onMouseMove(mouseEvent:MouseEvent):void {
      var charInText:int = tf.getCharIndexAtPoint(10, tf.mouseY);
      if (charInText == -1) return;
      var lineIndex:int = tf.getLineIndexOfChar(charInText);
      var firstCharIndex:int = tf.getLineOffset(lineIndex);
      var lastCharIndex:int = firstCharIndex + tf.getLineLength(lineIndex);

      //we want to set all the TextField to black except
      //the text underneath the mouse
      tf.setTextFormat(BLACK);
      tf.setTextFormat(RED, firstCharIndex, lastCharIndex);
    }
    protected function onMouseOut(mouseEvent:MouseEvent):void {
      tf.setTextFormat(BLACK);
    }
  }
}
```

First you use getCharIndexAtPoint() to convert a location on the screen to a location in the source text. I'll cover this next. Once you have an index into the text, you expand it to the whole line

by figuring out what line that character lies in, finding out the first character in that line, and finding the length of the line, from which you can easily derive the last character in the line. After you know where the line starts and ends in terms of character indices, you know enough to apply the formatting to the line.

Finding Text by Location

To map from on-screen positions to text indices, you can query a `TextField` by locations relative to its origin. Two methods are available:

- `getCharIndexAtPoint(x:Number, y:Number):int`
- `getLineIndexAtPoint(x:Number, y:Number):int`

The difference should be clear. Example 17-11 shows `getCharIndexAtPoint()` in use. Note that it uses the `y` position of the mouse relative to the `TextField`. Because the `TextField` is at the origin in this example, it's all the same, but remember that all locations are calculated with respect to the origin of the `TextField`.
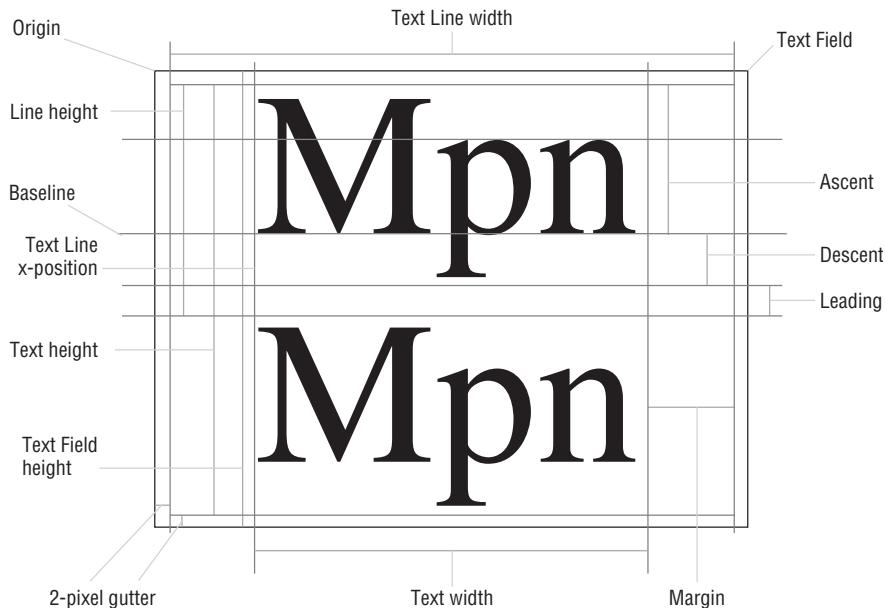
Locating and Measuring Text

You can get a lot of information about how specific lines of text are rendered in a `TextField` with the `getLineMetrics()` method. Given a line number, this method returns a `flash.text.TextLineMetrics` object populated with all kinds of information about the line of text (these measurements are illustrated in Figure 17-1):

- `x` — The left position of the first character in pixels
- `width` — The width of the text of the selected lines (not necessarily the complete text) in pixels

FIGURE 17-1

How text metrics measure a character and font

- `height` — The height of the text of the selected lines (not necessarily the complete text) in pixels
- `ascent` — The length from the baseline to the top of the line height in pixels
- `descent` — The length from the baseline to the bottom depth of the line in pixels
- `leading` — The measurement of the vertical distance between the lines of text

You can also convert text locations to on-screen locations for any given character with the `TextField` method `getCharBoundaries()`. Given a character's index in the source text, it will find its position and dimensions on the screen (relative to the origin of the `TextField` instance). This more direct mapping makes sense for some applications.

One great application of text measurement is to break up `TextFields` into separate fields for every word or letter, as Example 17-12 shows. This way, you can animate the words or letters separately to make text art.

EXAMPLE 17-12   `http://actionscriptbible.com/ch17/ex12`

Separating Text

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.geom.Rectangle;
  import flash.text.TextField;
  import flash.text.TextFormat;
  import flash.utils.setTimeout;
  public class ch17ex12 extends Sprite {
    protected var letters:Vector.<LetterField> = new Vector.<LetterField>();

    public function ch17ex12() {
      var text:String = "Are you watching closely?";
      var sourceTF:TextField = newTF();
      sourceTF.text = text;
      sourceTF.border = true;
      sourceTF.width = sourceTF.textWidth;
      addChild(sourceTF);

      //break it up into letters
      for (var i:int = 0; i < text.length; i++) {
        var lf:LetterField = new LetterField(newTF());
        var letterBounds:Rectangle = sourceTF.getCharBoundaries(i);
        lf.x = letterBounds.x;
        lf.y = letterBounds.y;
        lf.letter = text.charAt(i);
        addChild(lf);
        letters.push(lf);
      }

      //remove original
      removeChild(sourceTF);
      stage.addEventListener(MouseEvent.CLICK, startAnimation);
```

continued

EXAMPLE 17-12   *(continued)*

```
      }
    protected function startAnimation(event:MouseEvent):void {
      for (var i:int = 0; i < letters.length; i++) {
        setTimeout(letters[i].startAnimation, i * 30);
      }
    }
    protected function newTF():TextField {
      var tf:TextField = new TextField();
      tf.defaultTextFormat = new TextFormat("_serif", 38, 0, false, false);
      tf.selectable = false;
      tf.width = tf.height = 50;
      return tf;
    }
  }
}
import flash.text.TextField;
import flash.display.Sprite;
import flash.events.Event;
class LetterField extends Sprite {
  protected var tf:TextField;
  protected const factor:Number = 1 - 0.08;
  public function LetterField(tf:TextField) {
    this.tf = tf;
    addChild(tf);
  }
  public function set letter(ltr:String):void {
    tf.text = ltr;
    tf.x = -tf.textWidth / 2;
    this.x -= tf.x;
  }
  public function startAnimation():void {
    addEventListener(Event.ENTER_FRAME, tick);
  }
  protected function tick(event:Event):void {
    this.alpha *= factor;
    this.rotationY = 180 * (1-alpha);
  }
}
```

The main measurement code happens in the highlighted `for` loop. The bounding box for each letter is retrieved and used to position a new `TextField` that contains only that specific letter. After you have this framework for chopping up text, the possibilities are endless.

One interesting application of text metrics is getting text to flow around arbitrary outlines. You can see an ActionScript solution to this problem I wrote in 2007 (`http://dispatchevent.org/roger/dynamic-text-wrapping-in-actionscript-3/`). You'll see how it works in Example 36-13 after an introduction to bitmap programming.

Scrolling Text

Plain TextFields are capable of scrolling their content. If you put more text than fits in the size of the TextField, it acts as a window into the full size of the text. You can always write code to pan the TextField instance behind a mask or use its scrollRect property, but you can also use the built-in scroll properties. The difference is that TextFields scroll one row and column at a time, so you won't have "smooth" scrolling.

The user can scroll selectable and input TextFields by dragging a selection below the visible contents of the TextField or by cursoring out of the visible area. You can also enable scrolling via the mouse wheel by setting the mouseWheelEnabled property to true.

You can control the scrolling of a TextField with these properties:

- scrollV — The current vertical scroll position measured in lines of text (1-based) from the top. In other words, the index of the topmost visible line.

- bottomScrollV — The line index of the bottommost line visible in the TextField. You can determine how many rows are visible by subtracting scrollV from this. Read-only.

- scrollH — The current horizontal scroll position measured in pixels from the left. In other words, how many pixels left the content is being shifted.

- maxScrollH, maxScrollV — The maximum possible values for scrollH and scrollV. Read-only.

Because the scrollH and scrollV properties are read-write, you can scroll the TextField by setting these. In Example 17-13, you create a document reader that scrolls automatically as the mouse nears the top or bottom of the stage. A dim progress bar shows how far you are into the document.

EXAMPLE 17-13   `http://actionscriptbible.com/ch17/ex13`

Scrolling

```
package
{
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.text.TextField;
  import flash.text.TextFormat;
  public class ch17ex13 extends Sprite {
    protected var tf:TextField;
    protected var progressBar:Sprite;

    public function ch17ex13() {
      tf = new TextField();
      var fmt:TextFormat = new TextFormat("_serif", 24, 0x303030);
      fmt.leading = 10;
      fmt.leftMargin = fmt.rightMargin = 10;
      fmt.indent = 20;
      tf.defaultTextFormat = fmt;
```

continued

359

```
      tf.multiline = tf.wordWrap = true;
      tf.mouseWheelEnabled = true;
      tf.height = stage.stageHeight;
      tf.width = stage.stageWidth;
      tf.text = "Loading...";
      addChild(tf);

      progressBar = new Sprite();
      progressBar.graphics.beginFill(0, 0.1);
      progressBar.graphics.drawRect(0, 0, stage.stageWidth, 60);
      progressBar.graphics.endFill();
      addChildAt(progressBar, 0);

      var loader:URLLoader = new URLLoader(new URLRequest(
        "http://actionscriptbible.com/files/alice-ch1.txt"));
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onLoadComplete(event:Event):void {
      tf.text = URLLoader(event.target).data;
      tf.addEventListener(Event.ENTER_FRAME, scrollTextField);
      stage.frameRate = 15;
    }
    protected function scrollTextField(event:Event):void {
      var margin:Number = 50;
      if (stage.mouseY < margin) tf.scrollV--;
      if (stage.mouseY > stage.stageHeight - margin) tf.scrollV++;
      progressBar.y = (stage.stageHeight - progressBar.height)
        * (tf.scrollV / tf.maxScrollV);
    }
  }
}
```

Fonts

Of course, Flash Player would be no good at displaying type if it weren't able to use decent fonts. In this section you'll learn about the different kinds of fonts, ways to embed fonts, and even load fonts at runtime.

Device Fonts and Embedded Fonts

Thus far, all the code in this chapter has been using *device fonts*: fonts installed on the end-user's system. Flash Player actually uses the operating system it's running on to find and render these fonts, so it has less control over their appearance. You've been using the three generic font aliases: `_sans`, `_serif`, and `_typewriter`. When you use these three special aliases, Flash Player finds a generic sans-serif, serif, or monospaced device font on the system to use. You can't be sure that text rendered with these aliases will look the same across computers, but you can be sure that it will render.

Note

Japanese systems have three more generic aliases: _ゴシック, _明朝, and _等幅, for gothic (rounded and readable), minchou (Ming-style, more calligraphic), and touhaba (monospaced) fonts. ∎

What I haven't mentioned yet is that, using device fonts, you can render text in any font installed on the user's system. You can even use the Font class (coming right up) to find out what fonts are available and pick from them at runtime. In Example 17-14, if you can find Comic Sans or a similar font on your computer, you display it; if not, you act disappointed.

EXAMPLE 17-14 `http://actionscriptbible.com/ch17/ex14`

Locating and Using a Font

```
package {
  import flash.display.Sprite;
  import flash.text.Font;
  import flash.text.TextField;
  import flash.text.TextFieldAutoSize;
  import flash.text.TextFormat;

  public class ch17ex14 extends Sprite {
    public function ch17ex14() {
      var tf:TextField = new TextField();
      tf.autoSize = TextFieldAutoSize.LEFT;
      addChild(tf);

      var fonts:Array = Font.enumerateFonts(true);
      var found:Boolean = false;
      for each (var font:Font in fonts) {
        if (font.fontName.match(/^comic\b/i) != null) {
          tf.defaultTextFormat = new TextFormat(font.fontName, 14);
          tf.text = "Don't you love this font, " + font.fontName + "?!";
          found = true;
          break;
        }
      }
      if (!found) {
        tf.defaultTextFormat = new TextFormat("_typewriter");
        tf.text = "Lucky you, no Comic Sans!";
      }
    }
  }
}
```

The second kind of font is an embedded font. The outlines of all (or a subset of) the glyphs in the font is available to Flash Player because the font has been embedded in the SWF at compile time, or it's been loaded at runtime. Either way, Flash Player can use its own text engine to render text in the font exactly as intended. There are lots of benefits to using embedded fonts. Most importantly, you

can be sure that text renders as you want in that font, on every computer. Beyond that, you may also enable anti-aliasing and subpixel anti-aliasing for embedded fonts (see more about anti-aliasing in the later section "Anti-aliasing"), giving you lots of control over the smoothness of text using these fonts. Finally, there are some quirky limitations on what you can do with device fonts. A few of these were mentioned in "Introducing TextFields" earlier in the chapter.

Version

These limitations are also dependent on Flash Player version. In Flash Player 9, device text could not be rotated, scaled non-uniformly, or faded with `alpha`. It could be masked if it were cached into a bitmap first. Using Flash Player 10, device fonts are rendered with OS-provided anti-aliasing, and can be faded with `alpha`. As of Flash Player 10.1, device text still cannot be rotated with rotation, however it may be rotated with `rotationZ`, since this caches the text to a bitmap before transformation. However, using the new Flash Text Engine, covered in Chapter 18, "Advanced Text Layout," device text can be rotated. The takeaway is this: if you need to animate a `TextField`, its fonts should be embedded. ■

The main drawback of embedded fonts is their size. Embedding the outlines of a whole font adds some weight to your compiled SWF size. Thankfully, the world is becoming so broadband that few people will complain over 50kb of fonts being added to the download size, so this is only an issue in special cases like banners, where you really have to watch filesize carefully. Also thankfully, all the font embedding methods allow you to choose subsets of the font to embed, saving some space.

Beginning in Flash Player 10, embedded fonts come in two flavors. The old format, and the Compact Font Format or CFF. CFF fonts are designed for use in the Flash Text Engine; they won't work in a `TextField`. So, I'll shelve them until next chapter when I get really deep with text rendering.

Managing Active Fonts and the Font Object

Fonts in Flash Player are represented by `flash.text.Font` objects. Furthermore, the `Font` class is used to manage these fonts.

In the previous example, you saw how to retrieve all available fonts using the static method `Font .enumerateFonts()`. Pass it `true` to include all device fonts; omit this parameter or use `false` to include only embedded fonts. This method returns an array of `Font` instances.

Each `Font` instance has three essential properties.

- `fontName` — The name of the font. This is the string you will use in CSS rules, `<font>` tags, and the `font` property of `TextFormat`s.

- `fontStyle` — The style of this font. Bold, italic, and bold+italic versions of the font must be embedded independently if you are to use any text in that font and style. All these embedded fonts will be instances of `Font` with the same `fontName` but different `fontStyle`s. This property is one of the constants `FontStyle.REGULAR`, `FontStyle.BOLD`, `FontStyle.ITALIC`, or `FontStyle.BOLD_ITALIC`.

- `fontType` — Whether the font is a device font, an embedded font, or a embedded CFF font. These values are defined as constants in `FontType`.

But the most important duty of `Font` is to register new fonts. If you load a SWF with an embedded font, you'll need to do this step before you can use the font in `TextField`s. Once the embedded font is registered with `Font`, it can be referenced by font name by `TextFormat`s and the like. Use the static method `Font.registerFont()` to enable a font for use by passing its `Class` reference.

Embedding Fonts

The method to embed fonts in a SWF depends on the tool you use to compile, but the general idea is the same. You need to create a class for each embedded font.

Flash Builder, Flex Builder, Flex SDK, mxmlc

In all these tools, whether you're building Flex or just ActionScript content, you can use the Embed metadata tag to embed a font. You can embed fonts from the system you're compiling on by font name, or you can point the compiler to an OpenType or TrueType font on your system. See the Flex documentation at `http://bit.ly/flex-using-embed` for details on the [Embed] tag.

In Example 17-15, you embed a system font and then use it in a `TextField`.

EXAMPLE 17-15 `http://actionscriptbible.com/ch17/ex15`

Embedding a System Font

```
package {
  import flash.display.Sprite;
  import flash.text.*;
  import flash.utils.describeType;

  public class ch17ex15 extends Sprite {
    [Embed(systemFont="sansserif", fontName="mySans",
           mimeType="application/x-font", advancedAntiAliasing="true")]
    protected var fontClass:Class;

    public function ch17ex15() {
      var tf:TextField = new TextField();
      addChild(tf);
      tf.width = stage.stageWidth;
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.wordWrap = true;
      tf.antiAliasType = AntiAliasType.ADVANCED;
      tf.defaultTextFormat = new TextFormat("mySans", 12);
      tf.embedFonts = true;
      var fontInstance:* = new fontClass();
      tf.text = describeType(fontInstance);
    }
  }
}
```

I wanted to verify that the class you get from the [Embed] tag extends Font, so that instantiating the class will get you a Font instance, so I made an instance of the embedded class and used its description as the contents of the TextField. If you try out this example, you'll see that it does indeed extend Font.

However, you don't need to register this font manually; when you embed fonts into the SWF, they come preregistered. You can enumerate the fonts to verify this, or simply see that the font in this example is working splendidly without the extra step.

Flash Professional

When you're using Flash Professional to embed fonts, you can simply put a dynamic TextField on stage or in the library with the correct font embedded. This makes the font available by the same name that it reads in the TextField (the name reported by Flash).

You can also add a Font symbol to the library, check Export for ActionScript (make sure to press the Advanced button if this is not visible), and set a class name, making sure that it extends Font. In this case, you can give the font your own name.

There are some advantages to using Flash to embed your fonts. First, unlike Flash Builder, it can embed PostScript and Type 1 fonts, so if you're stuck with a font only in those formats, I suggest embedding it in a SWF using Flash, even if you don't use Flash to compile your project. You can load the font dynamically using pure ActionScript, and if you just embed once in Flash, you'll never have to worry about sharing the actual font between different computers, where bad things can happen (Mac and PC versions of the font embedding slightly differently, different versions of a common font by different foundries, different OSs naming the font differently…). Second, it's much easier to set ranges of characters to embed using Flash's interface than by staring at a Unicode chart. Personally, I use Flash Professional to embed all my fonts, sounds, and graphics, but Flash Builder to do all the programming and compiling, so these kinds of hybrid workflows, taking advantage of the strengths of each tool, are possible.

Loading Fonts Dynamically

This is one of those things that, if you mention it to someone who used ActionScript before ActionScript 3.0, they'll curl up into the fetal position and ignore you until you go away. Thankfully, with the Font class, all it takes is one simple method. Getting the Class reference to pass to Font.registerFont() is the hard part.

First, use a Loader to load the SWF that contains the font or fonts you want to embed. Once the SWF is done loading, you can pull class definitions out of it by the names of the classes.

```
var domain:ApplicationDomain = loader.contentLoaderInfo.applicationDomain;
var fontClass:Class = domain.getDefinition("assets.MyFontName") as Class;
Font.registerFont(fontClass);
```

Again, if you happened to instantiate the fontClass, you'd get a Font subclass.

Using Embedded Fonts

After you've got them registered, using embedded fonts is as easy as using the name of the font (as specified while exporting) in a TextFormat, CSS rule, <font> tag, or so on. Keep in mind that you have to have a separate copy of a font for each style: regular, bold, italic, and bold italic. You should also turn on the embedFonts property of any TextFields you'll be using embedded fonts in.

In Example 17-16, you'll load and use a font I've embedded using Flash Professional using the name "Inconsolata" and the class linkage com.actionscriptbible.assets.Inconsolata.

EXAMPLE 17-16   http://actionscriptbible.com/ch17/ex16

Dynamic Font Loading

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.net.URLRequest;
  import flash.system.*;
  import flash.text.*;
  public class ch17ex16 extends Sprite {
    protected const URL:String =
      "http://actionscriptbible.com/files/inconsolata.swf";
```

```
    protected const CLASS_NAME:String =
      "com.actionscriptbible.assets.Inconsolata";
    protected var loader:Loader;

    public function ch17ex16() {
      loader = new Loader();
      loader.load(new URLRequest(URL), new LoaderContext(
        true, null, SecurityDomain.currentDomain));
      loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onLoadComplete(event:Event):void {
      var domain:ApplicationDomain = loader.contentLoaderInfo.applicationDomain;
      var fontClass:Class = domain.getDefinition(CLASS_NAME) as Class;
      Font.registerFont(fontClass);

      var tf:TextField = new TextField();
      addChild(tf);
      tf.defaultTextFormat = new TextFormat("Inconsolata", 64);
      tf.antiAliasType = AntiAliasType.ADVANCED
      tf.width = stage.stageWidth; tf.height = stage.stageHeight;
      tf.multiline = tf.wordWrap = true;
      tf.embedFonts = true;
      tf.text = "Hello world";
    }
  }
}
```

The LoaderContext and its SecurityDomain settings are necessary for the online version of the example to load the font across domains.

Anti-aliasing

Anti-aliasing smoothes out rough edges of graphics by sampling the graphics at a higher resolution than your screen, averaging the results to turn some pixels neither fully on nor fully off, as shown in Figure 17-2. Starting in Flash Player 10, device fonts can use the host OS's text anti-aliasing technique when drawn on-screen. Otherwise, Flash Player's text rendering engine can draw embedded fonts with two major anti-aliasing modes, which can be further customized.

FIGURE 17-2

Comparing anti-aliased and bitmap fonts

Aliased (Bitmap)

Anti-aliased

Flash Player's text anti-aliasing comes in two varieties: normal and advanced. Advanced anti-aliasing uses subpixel anti-aliasing on your LCD monitor (you won't see the benefit with a CRT display) to get even smoother text at the expense of some slight color cast. It is slower overall, and it's more effective on smaller type sizes. Normal anti-aliasing uses only full pixels and ends up looking a bit heavier and softer, although these may be tweaked. It is speedier and looks smoother when animated, so it is recommended for animation.

Set the type of anti-aliasing to apply to text in the `TextField`'s `antiAliasType` property. Acceptable values include `AntiAliasType.NORMAL` and `AntiAliasType.ADVANCED`.

You can tweak some of the anti-aliasing settings, as well, if you select advanced anti-aliasing.

Fitting Edges to a Grid

The type of grid fitting used determines whether Flash Player forces strong horizontal and vertical lines to fit to a pixel or subpixel grid, or not at all. When you snap edges to pixel boundaries, you get increased sharpness at lower sizes at the expense of faithful reproduction. You can change the grid fitting algorithm by applying these values to the `gridFitType` property of a `TextField`:

- `GridFitType.NONE` — Doesn't attempt to fit the font to a grid. Good for larger sizes and faithful reproduction of glyphs.
- `GridFitType.PIXEL` — Strong horizontal and vertical lines are fit to a pixel grid. Gives sharp edges. This setting works only for left-aligned text fields.
- `GridFitType.SUBPIXEL` — Fits vertical edges to subpixel edges. Each pixel of your LCD is actually three subpixels jammed close together; fitting to this grid is a good balance between sharp edges using grid fitting and accurate reproduction.

Sharpness and Thickness

Using advanced anti-aliasing, you can also change the overall sharpness and thickness of the glyphs in a `TextField` by using the `sharpness` and `thickness` properties. Sharpness may be adjusted in a −400 to 400 range, and thickness in a −200 to 200 range. For both, 0 is the default.

Summary

- Use a `TextField` wherever you need to display text.
- The `TextField` is either dynamic or input.
- The `TextField` contains properties to control text wrapping and automatic sizing.
- `TextField` extends the `InteractiveObject` and `DisplayObject` classes and has all the properties, methods, and events inherited from them.
- `TextField` objects that are multiline have scroll properties that allow you to scroll up and down through the lines in the `TextField`.
- You can use a `TextFormat` object to control the appearance of text in the `TextField`.
- If the `TextField` has `htmlText`, you can use a `StyleSheet` object to apply CSS classes to HTML.
- `TextField` objects dispatch distinct `TextEvents` to indicate when the user has entered text if the `TextField` is an input field, or a link event if the `TextField` has `htmlText` with links within it.

Advanced Text Layout

The `TextField` object explored in Chapter 17, "Text, Styles, and Fonts," allows you to place text in your Flash applications. Some applications that deal extensively in text call for a deeper control over text than `TextField`s can provide. Supplementing, but not replacing `TextField`, Flash Player 10 adds a whole new text engine, called the Flash Text Engine. This engine provides comprehensive low-level control over the presentation of text. In addition, an open source framework built on this engine is provided: the Text Layout Framework. This framework lets you easily include print-quality text and layout tools in your application.

The new text engine provides functionality for letting text flow through connected text blocks like a prepress publishing tool, including columns and pages. It allows for the insertion of graphical elements into text, including text-wrapping options. It gives you the ability to lay out text vertically and include horizontal text within vertical text. It also provides higher levels of typographical control, over ligatures, digit case, kerning, hyphens, and so on. In this chapter you'll see all the specific applications of this toolkit, which are too numerous to list here.

FEATURED CLASSES

```
flash.text.engine.*

flashx.textLayout.*

flashx.undo.*
```

Version

FP10. The Flash Text Engine is available only in Flash Player 10 and later, as is the Text Layout Framework. I cover these technologies here, so this entire chapter is specific to Flash Player 10 and later. ■

Understanding Advanced Text Controls

The organization and use of the advanced text layout APIs is different from much of what's been covered in this book, so it merits a short overview. Up until this point, I've been covering functionality provided by classes and groups of classes in the Flash Player API. For example, to use text with a `TextField`, all that is

required is that you import and use the `TextField` class. The advanced text controls covered in this chapter, however, have been split into two parts.
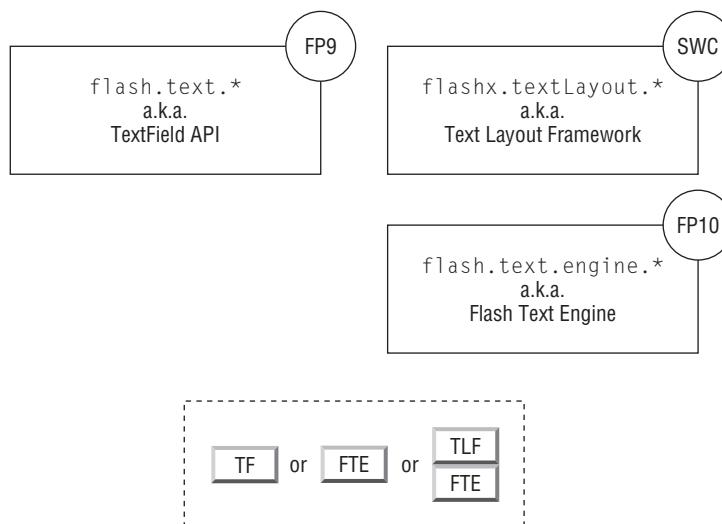
The Flash Text Engine, or FTE, is an addition to Flash Player 10. Like you've seen throughout the book, the capabilities of Flash Player are made available to programmers through classes and functions in the Flash Player API. The Flash Text Engine is available to programmers through the classes in the `flash.text.engine` package. Using these classes, you can take advantage of all the new capabilities of the text engine.

If you spent a long time writing ActionScript code, you could build a nice text layout library using the new Flash Text Engine that makes it easy to control the flow, layout, and presentation of text. Fortunately, Adobe has written an ActionScript library that does just this, the Text Layout Framework, and provided it for your use. Like any library, you can use this library by either including its source in your project or linking in precompiled SWC files. When using the Text Layout Framework, or TLF, you will use its API, housed in the `flashx.textLayout` package instead of the Flash Player's TFE API.

In summary, you have a choice. You can use an ActionScript library that gives you a higher-level and simpler interface (the Text Layout Framework), or you can use the core functionality that Flash Player gives you (the Flash Text Engine) directly to get more control. Or you can use a combination of both. Oh, and there's nothing that keeps you from mixing in the "legacy" text control, `TextField`. The Text Layout Framework is similar to the Flex framework: it uses ActionScript code to build on Flash Player and give you, the user of the library, powerful and customizable tools that would take time to build yourself. And, just like Flex, because the TLF is built on ActionScript code that is freely provided, you can extend it or override some of its behavior by interacting with its classes. Figure 18-1 shows the ways you can add dynamic text to your Flash application.

FIGURE 18-1

Three text APIs and where they are found. At bottom, the three options: you can't use TLF without FTE.

Advanced Text Controls in Development Tools

If you use Flash Professional, you can create TLF-powered text interactively. In Flash Professional CS5, the Text tool can create TLF text fields; the Properties panel lets you choose which text engine to use ("TLF Text" versus "Classic Text") and gives you access to the wide array of formatting options that the Flash Text Engine enables. If you use Flash Professional CS4, you can download the Text Layout Component from Adobe Labs, although it is deprecated and likely won't be maintained. The extension adds a TextLayout component to the Components panel — drag this to the stage to create a new text view — and a Text Layout panel, found in Other Panels, to edit the text content with all the options that FTE enables.

The new Spark text components in the Flex 4 framework are also built on FTE and TLF and support their advanced text features. These include `Label`, `RichText`, and `RichEditableText`. For Flex-specific information, see the Flex Language Reference or a Flex 4 text.

Why There Are Two Engines

It's true — the options for dealing with text can be daunting. The `TextField` API evolved over generations of Flash Players, gaining a whole lot of functionality organically from successive enhancements. I like to think of Flash Player 9 as the first version of the API because it is the first runtime for ActionScript 3.0. You can safely ignore anything that happened in Flash Player 8 and earlier, because those features are utterly incompatible with ActionScript 3.0. But the truth is, the `TextField` API covered in Chapter 17 is not much different from the ActionScript 2.0 `TextField`. It carries a lot of baggage, and its abilities were already stretched to the limit.

The Flash Player team put in a whole new text engine that's geared for the future. It has more power, more control, more structure, and more extensibility. If the developers had done this before releasing Flash Player 9, they could have removed `TextField` entirely, and you'd only need the Flash Text Engine. You'd never know what's missing, and books like this one wouldn't bother covering `TextField` because it would be an "ActionScript 2.0 thing," completely irrelevant to ActionScript 3.0.

But because Flash Player engineers added the Flash Text Engine *after* the tabula rasa of Flash Player 9, you will forever (until ActionScript 4.0?) have two text engines at your disposal. Pithily, because `TextField` is here, it's here to stay. Removing it would break all programs built for Flash Player 9 that use text.

The Flash Text Engine is the way of the future. Adobe is likely to make enhancements to FTE and TLF in future Flash Players, not `TextField`. It has integrated FTE into the new generation of Flex components. That said, at this moment, `TextField` is still fast and does a respectable job despite its idiosyncrasies. There's no shame in using it.

Where to Go from Here

I've just introduced a low-level Flash Player API (FTE) and a high-level ActionScript 3.0 library (TLF). The subject of this book is the Flash Player API, but its objective is to be useful. I'll spend a majority of this chapter on the Text Layout Framework, because it's the more powerful tool that you'll use in real projects. This higher-level framework gives you all the control you'll need over text in all but the rarest cases, and it does so with less coding effort required.

The Flash Text Engine
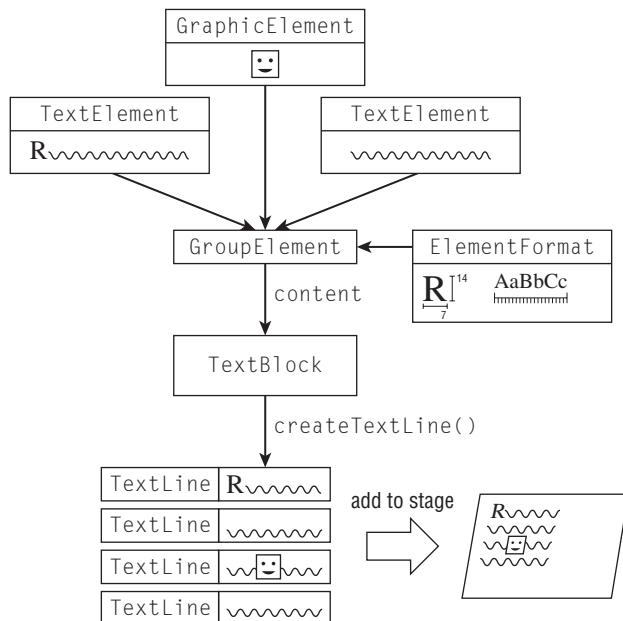
Even though I'll be focusing on the more convenient Text Layout Framework, I'll quickly get you acquainted with the new text engine it's based on. This will help you understand how the TLF works, and you may find occasion to use the FTE on its own.

The ultimate purpose of the FTE is to properly format and display a set of lines of text. Whereas the operation of `TextField` is opaque to ActionScript — you give it text and a format or CSS that entirely determine its appearance — the FTE exposes its inner workings down to the line level, and then it can give you some more information about individual letters or "atoms" if you request it. Its workings are open to you because, rather than a single class, the FTE consists of a set of classes that work together. Rather than simply assign content and formatting to a `TextField`, you break down your content into groups of `ContentElements` such as `TextElement` and `GraphicElement`, give each one of these an `ElementFormat`, and provide the `ContentElements` to a `TextBlock`, which is more or less a paragraph. Then you call the `createTextLine()` method of your `TextBlock` repeatedly, and the `TextBlock` figures out how the text should flow and look line by line, generating a display object, `TextLine`, as the result of each call. This hands-on process gives the opportunity to exercise detailed control, or at minimum to see what's happening. See the classes involved in the FTE in Figure 18-2.

FIGURE 18-2

Flash Text Engine architecture

Why is the end product of this hands-on process a line of text? Why not a character, or the entire paragraph? The engine, like a print layout program, has to make some tough decisions to figure out what can go into a line. It must consider the sizes of the content it has to lay out, what content it has already laid out, the available dimensions for a line, the justification of the text, the natural places text can break, the rotation of the content, and so on. The font and formatting properties like kerning and tracking do most of the work to determine how letters are laid out within the line. In general, a line of text is the level of detail that's going to be most useful to applications looking to compose any moderate quantity of text on the screen. Specifically, the TLF benefits from line-level control as this allows it to easily flow text from one text container (a column, for instance) to another — if the last line fills up the container, the next line should go in the next container. Rarely do you make layout decisions based on a single character.

This hands-on interface roughly follows a model-view-controller architecture. There is only one view in the FTE — only one class that extends `DisplayObject` — and that's `TextLine`, which contains a single line of text. It extends `DisplayObjectContainer`, so it dispatches the normal `InteractiveObject` events. It can provide measurements about the text that it displays. However, a `TextLine` doesn't operate in a vacuum: it's created by, owned by, and managed by a `TextBlock`. You can't even create a `TextLine` in ActionScript except by using `TextBlock`: the `TextLine()` constructor throws an error.

The `TextBlock` class is your controller for a given set of content. You can consider it your handle to the FTE. It owns the content, the formatting, and the views for a given block of text. You can think of this as the workhorse of the FTE. Once you provide it with the content and formatting, it can figure out how that content is broken into `TextLines`. Furthermore, it owns all created `TextLines` and even revises them as necessary. If you change the content those lines are displaying, the `TextBlock` controller marks them invalid and reflows the text as necessary.

Note
I sometimes use the word "content" instead of "text" because the FTE can lay out both graphics and text. ∎

While the controller and view are solitary, the FTE has many model classes, which specify both the content to be displayed and the formatting to be applied, including the properties of the font used. The kinds of content are modeled by `ContentElement` subclasses: `GroupElement`, `GraphicElement`, and `TextElement`. Using the `GroupElement` class, you can sequence and nest runs of text and inline graphics. This is especially important considering that each `TextElement` has a single format model that applies to the entire run of text. By using a `GroupElement`, you can string together runs of text with separate formats. The FTE uses another model class for specifying these formats: `ElementFormat`. This class specifies all the properties of a given content element, like its font, color, font size, kerning, tracking, typographic case, relative rotation, and disposition toward using ligatures and breaking onto a new line. It uses a further model class, `FontDescription`, to detail how the intended font family, weight, and posture should be located and rendered. These models work together to fully describe the content before it's broken into discrete lines and rasterized. The models by themselves do nothing, just like the view by itself does nothing. Everything hinges on how you use the `TextBlock` controller to convert content and formatting models into display objects containing the content.

Let's see how the models, view, and controller collaborate to layout and display text using the Flash Text Engine, in Example 18-1.

Using the Flash Text Engine

```
package {
  import flash.display.Sprite;
  import flash.text.engine.*;
  public class ch18ex1 extends Sprite {
    private const LEADING:Number = 4;
    private const FONT_SIZE:Number = 14;
    private const FONT_COLOR:Number = 0x303070;
    public function ch18ex1() {
      //create the content and formatting models, and associate them.
      var content:TextElement = new TextElement("Lorem ipsum dolor sit amet, \
consectetur adipiscing elit. Nulla massa magna, lobortis non viverra a, \
hendrerit ut est. Nullam eget mauris ac nisl iaculis scelerisque. \
In sodales orci posuere sem gravida luctus.\n\n\
Did I mention Hello World in there?");
      var fontDescription:FontDescription = new FontDescription("_typewriter");
      content.elementFormat =
        new ElementFormat(fontDescription, FONT_SIZE, FONT_COLOR);

      //create the controller, and give it the model.
      var block:TextBlock = new TextBlock();
      block.content = content;

      //use the controller to generate views, and add them to stage.
      var line:TextLine = null;
      var textY:Number = 0;
      while (line = block.createTextLine(line, stage.stageWidth)) {
        textY += line.textHeight + LEADING;
        line.y = textY;
        addChild(line);
      }
    }
  }
}
```

Clearly, creating a simple block of text directly with the FTE requires a lot of set-up code. But this is the price you pay for lower-level access to the text facilities. You could investigate further the text applications possible with just the FTE, but instead let's see what's possible with the higher-level Text Layout Framework. As you see what the TLF can do for you, it will become clear how the TLF leverages the abilities of the FTE that you've become acquainted with here.

The Text Layout Framework

The Text Layout Framework enables you to easily display print-quality text, with the features you'd expect from a prepress layout tool, all at runtime inside Flash Player 10 and later. As you've learned, it's an open source library written in ActionScript 3.0 and leveraging the FTE; if you like, you

can open, view, and even edit the source. Include the TLF in your Flash application by adding its precompiled SWCs to your project. The TLF is included with the Flex 4 framework (included in Flash Builder 4 and later) and with Flash Professional CS5 and later. Otherwise, you can find source and SWCs at `http://bit.ly/text-layout-framework`. Classes that make up the TLF are found in the package `flashx.textLayout`. The `flashx` package will presumably be used for other noncore (external to the Flash Player API) Adobe extensions that do not require Flex.

Note

SWCs are linked in by the compiler. Flash Professional, Flash Builder, and the Flex SDK compilers use separate methods to link SWCs when compiling. Flash Professional puts these options in the File ⇨ Publish Settings ⇨ Flash ⇨ ActionScript Settings ⇨ Library Path dialog box. Flash Builder lets you configure linking in the Project ⇨ Properties ⇨ ActionScript Build Path ⇨ Library Path dialog box. The `mxmlc` command-line compiler uses the `-include-libraries` or `-runtime-shared-library` switches to link in Runtime Shared Libraries (RSLs).

The TLF SWC is a signed Adobe framework, and as such, it can be cached by Flash Player when you choose to link in the SWC as an RSL. I recommend that you take advantage of framework caching when using the TLF. Read more about framework caching at `http://bit.ly/framework-rsl` ∎

So what does using the TLF get you? Besides the advanced capabilities that the FTE provides — ligatures, typographic case, digit case and width, DefineFont4 font embedding, right-to-left and vertical languages, tate-chu-yoko text, tab setting, inline graphics, and so on — the TLF adds new features while making the FTE's features much easier to use. The major features fall into three categories:

- Text Containers — Although the FTE's ultimate goal is to produce lines of text, it leaves laying them out to us. The TLF takes care of the layout, giving you flexible ways to flow text between text containers, around graphics, into columns, and so on. Rather than being used to lay out a single paragraph like `TextBlock`, the FTE lays out any length of content in any number of paragraphs or pages.

- Content Markup — Creating the proper `ContentElement` and `ElementFormat` objects, and associating them, is a painful chore to do by hand. The TLF defines a markup language, which maps to richer Element objects that it defines. It also provides facilities to import and export formatted content to markup, plaintext, or the kind of simplified HTML that `TextField` supports.

- Editability and Events — TLF adds support for text selection, including naturally selecting text across multiple container boundaries; cut, copy, paste, and clipboard control; and link events.

I'll cover these new features and dive into code after explaining the organization of the framework.

Storing Content and Formatting with Models

The Text Layout Framework is also based on the model-view-controller design pattern. Whereas the core function of TFE is to produce lines of text, the purpose of the TLF is to flow a longer body of text, generically called a *story*. The naming and organization of the classes that participate in each technology reflect their purpose.

You can also break up TLF models into content and formatting models. The content models, found in the package `flashx.textLayout.elements`, extend from the abstract `FlowElement` class and are organized hierarchically into groups, much like the structure of FTE content elements. However, TLF content is far more structured and organized. The different kinds of TLF content elements, perhaps inspired by HTML elements, help you organize sections of text and inherit formatting efficiently. (Recall that all `ContentElements` in the TFE require their own `ElementFormat`, and there is no inheritance of styles.)

This more complex hierarchy can be drawn in a tree, in which the root node is a `TextFlow`, interior nodes are subclasses of `FlowGroupElement`, and leaf nodes are subclasses of `FlowLeafElement`. Figure 18-3 shows such a structure, which also introduces all the kinds of elements that you can use in a story. The root node is always a `TextFlow` element. This can contain `DivElements`, which group paragraphs with the same style. `ParagraphElements` are single paragraphs, which are in turn composed of spans of text (`SpanElement`), inline images (`InlineGraphicElement`), links (`LinkElement`), and tate-chu-yoko text (`TCYElement`). Note that divs can be nested, but paragraphs cannot. You can think of this hierarchy as similar to the HTML tags:

```
<body>
  <div>
    <p>
      <span></span>
      <img />
      <a></a>
    </p>
  </div>
</body>
```
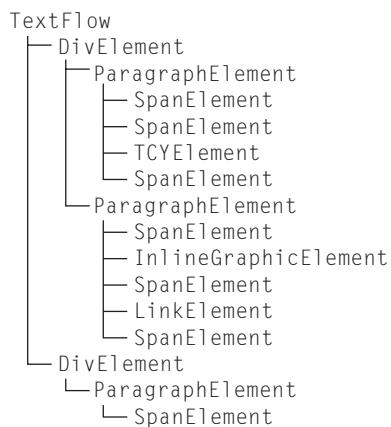
Note

Japanese text can be written horizontally, left-to-right, or vertically, top-to-bottom and right-to-left. When vertical Japanese text includes a short bit of non-Japanese text, like a number or acronym, it may be rotated to appear horizontally, making the horizontal text legible in its natural orientation, without breaking the overall vertical flow of text. This is tate-chu-yoko, or "horizontal within vertical" text. Now you know. ■

FIGURE 18-3

A hierarchy of `FlowElements`, using all possible concrete `FlowElement` types

```
TextFlow
├─ DivElement
│  ├─ ParagraphElement
│  │  ├─ SpanElement
│  │  ├─ SpanElement
│  │  ├─ TCYElement
│  │  └─ SpanElement
│  └─ ParagraphElement
│     ├─ SpanElement
│     ├─ InlineGraphicElement
│     ├─ SpanElement
│     ├─ LinkElement
│     └─ SpanElement
└─ DivElement
   └─ ParagraphElement
      └─ SpanElement
```

It should be no surprise that this tree structure is easily written using XML. In Text Layout markup, which I'll cover in detail shortly, tags map to element classes, and attributes map to properties of the class. This mapping is familiar from Flex's MXML and other languages that serialize runtime objects to XML (in the vein of the Inversion of Control/Dependency Injection design pattern). It also looks a bit like HTML, but you'll see that it's very different.

Formatting inherits from parent elements up the tree, and each `FlowElement` can override individual formatting properties of its parents, much like cascading styles. This is far more complex than the simple each-element-has-a-format-object method of the FTE, but it's still a still far cry from the rich selectors language that CSS provides. In fact, setting and inheriting styles are a principal motivation for separating your content into divs and spans.

Conveniently, `FlowElements` can specify a given formatting property directly on the object rather than accessing a format object. All the text formatting properties are defined by the `ITextLayoutFormat` interface, which `FlowElement`, the base class for all content nodes, implements. Interestingly, there is also a `TextLayoutFormat` object that implements this interface. This confusing fact provides parallel ways to set formatting on a given flow element: simply set the appropriate property on the element object itself, or assign it to a `TextLayoutFormat`.

```
var textFlowElement:TextFlow = new TextFlow();

//assign format by setting property
textFlowElement.color = 0xff0000;

//assign format by setting TextLayoutFormat object
//note that all property names e.g. "color" are identical
var newFormat:TextLayoutFormat = new TextLayoutFormat();
newFormat.color = 0xff0000;
textFlowElement.format = newFormat;
```

Overriding a single formatting property might be achieved better through the `ITextLayoutFormat` property of a `FlowElement` — for instance, adding italic to a foreign phrase — whereas setting an overall, reusable style is better accomplished by saving the format as a `TextLayoutFormat` and assigning it to the format property of any `FlowElement` that you want to appear that way — such as reusing a block quote style for several quotes scattered through a story. You'll see both methods reflected in TLF markup shortly.

Laying Out Text with Controllers

Okay, so you can write and format text, but this is pointless until you can put it on-screen. Ultimately, the TLF generates `TextLines` just as the FTE does. But the TLF can do more — namely, flowing text across connected containers. Again, the controller classes do this work, consuming the models and producing views. In the Text Layout Framework, you have the option of using a simple controller that merely generates lines, or one that composes the lines within and through multiple containers.

Simple Composition with TextLine Factories

One kind of controller for TLF text is the `TextLine` factory. There are two of these, both of which inherit from `TextLineFactoryBase`, found in the `flashx.textLayout.factory` package. `TextLine` factories produce and arrange `TextLines` within a simple rectangular area, although they stop just short of adding them to the display list for you. This is already more than the FTE did for you; recall that in Example 18-1 you had to write your own loop to generate the lines, and then you arranged them yourself vertically. The `TextLine` factories take into account the writing direction and styles like leading when laying out these lines.

In Example 18-2, you'll take a model — remember, the root element of all content must be a `TextFlow` — and use the factory to generate lines out of it.

Using the TextLine Factory

```
package {
  import flash.display.*;
  import flash.geom.Rectangle;
  import flash.text.engine.LigatureLevel;
  import flashx.textLayout.elements.*;
  import flashx.textLayout.factory.TextFlowTextLineFactory;
  import flashx.textLayout.formats.TextLayoutFormat;
  public class ch18ex2 extends Sprite {
    public function ch18ex2() {
      //create the content models TextFlow > Paragraph > Span > text
      var content:TextFlow = new TextFlow();
      var p:ParagraphElement = new ParagraphElement();
      var span:SpanElement = new SpanElement();
      content.addChild(p);
      p.addChild(span);

      span.text = "Some say the world will end in fire,\n\
Some say in ice.\n\
From what I've tasted of desire\n\
I hold with those who favour fire.\n\
But if it had to perish twice,\n\
I think I know enough of hate\n\
To say that for destruction ice\n\
Is also great\n\
And would suffice.\n\n\
\u2014Robert Frost"; //unicode for em dash used in this line.

      //create the format model. use ligatures and leading
      var format:TextLayoutFormat = new TextLayoutFormat();
      format.fontFamily = "Caslon, Garamond, _serif";
      format.ligatureLevel = LigatureLevel.COMMON;
      format.fontSize = 17;
      format.lineHeight = 28;
      format.color = 0x303030;
      content.format = format;

      //create a factory, set its bounds, and let it generate lines
      var controller:TextFlowTextLineFactory = new TextFlowTextLineFactory();
      var fullW:Number = stage.stageWidth;
      var fullH:Number = stage.stageHeight;
      controller.compositionBounds = new Rectangle(fullW/4, 20, fullW/2, fullH);
      controller.createTextLines(onLineCreated, content);
    }
```

```
    //called by the TextFlowTextLineFactory every time a line is created
    private function onLineCreated(line:DisplayObject):void {
      //it's still our job to add these to the display list
      addChild(line);
    }
  }
}
```

First you created the controller with its empty constructor. You defined the area that text will be added to by setting the factory's compositionBounds property. Finally, you called createTextLines(), passing it a callback and a TextFlow, which makes up the story's content. The factory lets you do what you please with the lines it creates, but they're already positioned and ready to go, if you want to just add them to the stage as you've done here. For a shortcut, try using a Function reference to addChild() as the callback function.

```
        controller.createTextLines(addChild, content);
```

Now you can remove the custom callback function onLineCreated().

There's an even simpler TextLine factory in the TLF that uses a flat string as content instead of a proper TextFlow object. Because Example 18-2 used a single format for a single span of text, it could easily be rewritten to use the StringTextLineFactory. Try modifying the example to use this, the simplest of the TLF controllers. How many lines of code did you save?

Besides the fact that TextLine factories lay out FTE text in a simple manner, they are also limited to static text. They don't support editing, real-time updates, or activities like selection, cut, or copy. That's not to say they're useless, though. The content and formatting objects can specify intricate details of the text, complex alignment, and even columns. Because columns are one of the big reasons you might use flowed containers, it's convenient that columns are built-in. Take a look at Example 18-3.

EXAMPLE 18-3 `http://actionscriptbible.com/ch18/ex3`

Using TextLine Factories with Column Layouts

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.geom.Rectangle;
  import flash.net.*;
  import flashx.textLayout.factory.StringTextLineFactory;
  import flashx.textLayout.formats.TextLayoutFormat;
  public class ch18ex3 extends Sprite {
```

continued

EXAMPLE 18-3 *(continued)*

```
  public function ch18ex3() {
    var loader:URLLoader = new URLLoader(new URLRequest(
      "http://actionscriptbible.com/files/alice-ch1.txt"));
    loader.addEventListener(Event.COMPLETE, onLoadComplete);
  }
  protected function onLoadComplete(event:Event):void {
    var controller:StringTextLineFactory = new StringTextLineFactory();
    controller.text = URLLoader(event.target).data;
    controller.compositionBounds =
      new Rectangle(5, 5, stage.stageWidth-10, stage.stageHeight-10);
    var format:TextLayoutFormat = new TextLayoutFormat();
    format.fontFamily = "Book Antiqua, Garamond, _serif";
    format.fontSize = 13;
    format.columnCount = 3;
    format.columnGap = 15;
    controller.textFlowFormat = format;
    controller.createTextLines(addChild);
  }
}
}
```

Example 18-3 shows the `StringTextLineFactory` used with a three-column layout. You can
see that it's as simple as setting a few properties on a format object. Because the text is loaded as a
`String`, it's a single block that you can apply only one style to. That's fine for your purposes here.
You also used the `addChild` callback trick. All in all, it's quite a compact way to use the TLF.

Linked Container Composition with ContainerControllers and FlowComposers

When you use linked text containers (or even a single container) for text, you can get a lot more out
of the TLF, including re-flowing text, editable text, text selection, scrolling, focus, and clipboard con-
trol. Once you've defined your content with a `TextFlow`, follow these general steps:

- Create a `Sprite` for each container. This will hold all the `TextLines` and other necessary
  `DisplayObjects` like inline graphics, background colors, and interactive feedback (highlight-
  ing a selection, for instance).

- Create a `ContainerController` for each container, associating it with the container
  `Sprite`. The controller is responsible for the contents of its own container. Essentially, it
  breathes life into the empty `Sprite`, endowing it with the various `TextField`-like abilities
  you take for granted.

- Create an `IFlowComposer` class and associate it with the model by assigning it to the
  `TextFlow`'s flowComposer property. The TLF comes with `StandardFlowComposer`,
  which implements this interface but leaves you the option of rolling your own. The flow
  composer manages all the connected containers, orchestrating their activity so that they act like
  a single continuous layout.

- Add the container controllers to the flow composer in the same order that text will flow through
  them. `IFlowComposer` defines a set of methods for managing its `ContainerControllers`
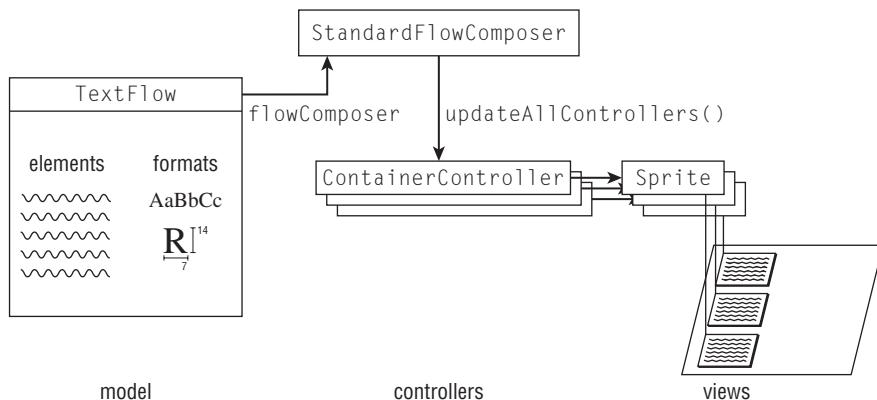
that recall the display list API, starting with `addController()`, `addControllerAt()`, and `removeController()`. (Consult the AS3LR for the full battery of methods.)

- Finally, when you want to render the story into its text containers, tell the `IFlowComposer` to `updateAllControllers()`.

The setup required here is nontrivial, but it makes sense. All the players in this game have to know about each other. The flow composer owns all the container controllers, each of which manage their own container sprites. The model knows who's composing the text so that when it changes, it can notify the flow composer that it needs to redraw. You can see how the classes interact in Figure 18-4.

Classes used in flowed text layout

These classes also act together to keep tabs on which parts of the text need to be redrawn. This way, the expensive process of figuring out what lines go where is minimized as well as localized. The TLF exposes access to these invalid or *damaged* zones if you need to flag them manually or check if they are damaged.

Because you already know how to use text in columns without flowed containers, I'll demonstrate flowed text with a more complex application in Example 18-4.

EXAMPLE 18-4   `http://actionscriptbible.com/ch18/ex4`

Flowed Containers

```
package {
  import flash.display.Sprite;
  import flash.geom.Point;
  import flash.text.FontStyle;
  import flashx.textLayout.compose.StandardFlowComposer;
  import flashx.textLayout.container.ContainerController;
  import flashx.textLayout.edit.SelectionManager;
  import flashx.textLayout.elements.*;
  import flashx.textLayout.formats.TextAlign;
```

continued

EXAMPLE 18-4 *(continued)*

```
  public class ch18ex4 extends Sprite {
    public function ch18ex4() {
      var textFlow:TextFlow = new TextFlow();
      var p:ParagraphElement = new ParagraphElement();
      var span:SpanElement = new SpanElement();
      span.text = "Shall I compare thee to a summer's day?\n\
Thou art more lovely and more temperate.\n\
Rough winds do shake the darling buds of May,\n\
And summer's lease hath all too short a date.\n\
Sometime too hot the eye of heaven shines,\n\
And often is his gold complexion dimmed;\n\
And every fair from fair sometime declines,\n\
By chance, or nature's changing course, untrimmed;\n\
But thy eternal summer shall not fade,\n\
Nor lose possession of that fair thou ow'st,\n\
Nor shall death brag thou wand'rest in his shade,\n\
When in eternal lines to Time thou grow'st.\n\
So long as men can breathe, or eyes can see,\n\
So long lives this, and this gives life to thee.\n\
\u2014William Shakespeare";
      p.addChild(span);
      textFlow.addChild(p);

      //create an IFlowComposer
      var composer:StandardFlowComposer = new StandardFlowComposer();
      //assign it to the story
      textFlow.flowComposer = composer;

      var COUNT:int = 14;
      var RADIUS:Number = stage.stageWidth * 0.7;
      var ORIGIN:Point = new Point(stage.stageWidth/2, stage.stageHeight - 20);
      for (var theta:Number = 0; theta <= Math.PI; theta += Math.PI/COUNT) {
        //create a sprite for each container
        var sprite:Sprite = new Sprite();
        sprite.x = -RADIUS * Math.cos(theta) + ORIGIN.x;
        sprite.y = -RADIUS * Math.sin(theta) + ORIGIN.y;
        sprite.rotation = theta / Math.PI * 180;
        addChild(sprite);
        //create a controller for each container, and add it to the composer
        composer.addController(new ContainerController(sprite, RADIUS-50, 20));
      }

      textFlow.textAlign = TextAlign.END;
      textFlow.fontFamily = "Garamond, _serif";
      textFlow.fontStyle = FontStyle.ITALIC;
      textFlow.fontSize = 14;
      textFlow.color = 0xd09000;
```

```
        textFlow.interactionManager = new SelectionManager();
        //instruct the composer to flow the text into its containers
        textFlow.flowComposer.updateAllControllers();
      }
    }
  }
```

This example sets Shakespeare's sonnet to the shape of sunrays. Although that's nothing you couldn't have done with `TextFields`, when you interact with flowed text like this, it's actually one contiguous block of text. I've demonstrated this by enabling selection on the whole sonnet. The text appears all over the screen, but you can still use the mouse to select text perfectly.

Text Layout Markup

Now that you understand the structure of the TLF and its model classes, you can start using markup in place of those classes. TLF markup language (TLFML? Let's not.) provides the same facilities as the content model, but in a compact, portable XML format. TLF markup is converted to TLF objects at runtime using a text importer class, and you can use a text exporter class to turn TLF objects back into TLF markup.

In markup, each of the content element objects gets its own XML element; nesting two tags creates the same parent-child relationship created at runtime with the elements' `addChild()` methods.

```
//TLF code
var div:DivElement = new DivElement();
var p:ParagraphElement = new ParagraphElement();
div.addChild(p);
//TLF markup
<div><p></p></div>
```

Attributes are used mostly for formatting properties, just as you can set formatting properties on `FlowElements`, because `FlowElement` implements `ITextLayoutFormat`.

```
//TLF code
var p:ParagraphElement = new ParagraphElement();
p.trackingRight = 3;
//TLF markup
<p trackingRight="3"></p>
```

Additionally, you can save a format and reuse it in various places like a CSS rule or an old-school `TextFormat`. Define the format with a `<format>` tag, name it with the `name` attribute, and reference it in any elements that you want it to apply to.

```
//TLF code
var red:TextLayoutFormat = new TextLayoutFormat();
red.color = 0xff0000;
```

```
var p:ParagraphElement = new ParagraphElement();
var s1:SpanElement = new SpanElement();
s1.text = "Mind your ";
p.addChild(s1);
var s2:SpanElement = new SpanElement();
s2.text = "p";
s2.format = red;
p.addChild(s2);
var s3:SpanElement = new SpanElement();
s3.text = "s and ";
p.addChild(s3);
var s4:SpanElement = new SpanElement();
s4.text = "q";
s4.format = red;
p.addChild(s4);
var s5:SpanElement = new SpanElement();
s5.text = "s!";
p.addChild(s5);
//TLF markup
<format name="red" color="0xff0000"/>
<p>
  <span>Mind your </span>
  <span format="red">p</span>
  <span>s and</span>
  <span format="red">q</span>
  <span>s!</span>
</p>
//TLF markup, simplified
<format name="red" color="0xff0000"/>
<p>Mind your <span format="red">p</span>s and <span format="red">q</span>s!</p>
```

TLF markup is so much more compact that you might want to use it even if you're not loading it externally, by using inline XML.

Caution

If you're using an older build of the TLF, the `name` property may be instead called `id`. You can cover your bases by using both attributes:

```
<format name="red" id="red" color="0xff0000"/>
```

Setting Up TLF Markup

All TLF tags and attributes are namespaced, so you need to include this namespace in any TLF markup you write. Set it to the default namespace for conciseness:

```
<TextFlow xmlns="http://ns.adobe.com/textLayout/2008">
  <format name="red" color="0xff0000"/>
  <p>Mind your <span format="red">p</span>s
    and <span format="red">q</span>s!</p>
</TextFlow>
```

382

Or, if you're combining it with other XML, you may want to use a namespace handle:

```
<tlf:TextFlow xmlns:tlf="http://ns.adobe.com/textLayout/2008">
  <tlf:format name="red" color="0xff0000"/> ...
```

Refer to Chapter 11, "XML and E4X," for more information on handling XML and namespaces in ActionScript.

TLF Tags

The TLF markup language is fairly simple. There are only a few defined elements, which are summarized in Table 18-1. The table is roughly ordered: elements that can be nested in a given tag mostly appear in rows below it.

TABLE 18-1

TLF Markup Tags

Element Name	Equivalent Class	Description	Allowed Children
`<TextFlow>`	`TextFlow`	The root element of a text flow	Text, any tag below this
`<format/>`	`TextLayoutFormat`	A reusable format	None
`<div>`	`DivElement`	A section or group of paragraphs	Text, any tag below this
`<p>`	`Paragraph Element`	A paragraph	Text, any tag below this
`<tcy>`	`TCYElement`	Tate-chu-yoko text	Text, any tag below this
`<a>`	`LinkElement`	A clickable link	Text, any tag below this
`<img/>`	`InlineGraphicElement`	Inline graphics	None
`<span>`	`SpanElement`	A range of text	Text, ` `, `<tab>`
` `	`BreakElement`	Newline character (\n)	None
`<tab/>`	`TabElement`	A tab character (\t)	None
`<linkNormalFormat>`	`TextLayoutFormat`	Overriding formats for a link	None
`<linkActiveFormat>`	`TextLayoutFormat`	Overriding formats for a link while being activated	None
`<linkHoverFormat>`	`TextLayoutFormat`	Overriding formats for a link while being hovered	None

I should mention a few things about TLF markup that are easy to miss from just looking at this table. First of all, notice that all the tags are lowercase, except for `<TextFlow>`. It's essential to get the case of your tags correct.

The `<br>` and `<tab>` tags, although they have corresponding elements, just turn into their respective characters in the text when imported. The link format tags override formatting properties for the link in the appropriate state, and they may appear almost anywhere. Their values cascade down the element tree like any formatting property.

You'll notice that TLF markup lets you elide purely structural elements. For example, you can add text directly to a `<TextFlow>` if you don't need the structure or style of the intermediate paragraph or span. During importing, a default `ParagraphElement` and `SpanElement` are added to the hierarchy.

As I've said, all the attributes of TLF tags are formatting properties. Some of these formats apply only to the block-level elements `<TextFlow>`, `<div>`, and `<p>`. For example, you can't set the justification of a single character. It only makes sense for a paragraph or set of paragraphs. Because they're so numerous, you can read about the available formatting properties in the AS3LR under the `ITextLayoutFormat` interface.

Importing and Exporting Markup

The `flashx.textLayout.conversion` package contains all you need to import and export TLF markup. The import/export capabilities of the TLF are particularly open to extension, if you care to define or create an adapter for another format. Two interfaces and their methods define all you need to convert text between a TLF `TextFlow` and another format.

`ITextImporter` imports text into `TextFlow` objects with the following method:

```
function importToFlow(source:Object): TextFlow
```

Additionally, `ITextImporters` expose any problems they may have run into during conversion with the property `errors`, a `Vector` of `String` error messages. You can instruct the importer to raise exceptions by setting its `throwOnError` property to `true`.

`ITextExporter` exports `TextFlow` objects to another format with this method:

```
function export(source:TextFlow, conversionType:String):Object
```

The method is intended to produce either XML or plaintext, which can be selected with the `conversionType` parameter. Use `ConversionType.STRING_TYPE` or `ConversionType.XML_TYPE`.

Unless you're creating your own format, you'll probably be happy to stick with the built-in class `TextConverter`, which can import and export in four formats. It's a bit quirky, but `TextConverter` doesn't implement either of the aforementioned interfaces (although it can return objects that do conform to those interfaces). Instead, it has similar static methods; these provide a few more options, and let you use the class without instantiating it.

```
function importToFlow(source:Object,
                      format:String,
                      cfg:IConfiguration = null):TextFlow
```

Use this method to convert your markup into a `TextFlow`. It's smart enough to know whether `source` is a `String` or XML object, so pass whichever you like. To indicate which of the supported

formats to use, pass its constant (such as TextConverter.TEXT_LAYOUT_FORMAT for TLF markup) to format. The optional cfg parameter sets an initial configuration for the TextFlow, which can always be set at a later time. You'll read about configuring a TextFlow in the section "Flow and container configuration."

```
function export(source:TextFlow, format:String, conversionType:String):Object
```

This method of TextConverter exports a TLF story to some other format. It also adds a format parameter, using the same formats and same constants to indicate them.

Example 18-5 should put TextConverter into context. You'll find that it's simple to use.

EXAMPLE 18-5   `http://actionscriptbible.com/ch18/ex5`

Using TextConverter

```
package {
  import com.actionscriptbible.Example;
  import flash.geom.Rectangle;
  import flash.text.*;
  import flashx.textLayout.conversion.ConversionType;
  import flashx.textLayout.conversion.TextConverter;
  import flashx.textLayout.elements.TextFlow;
  import flashx.textLayout.factory.TextFlowTextLineFactory;
  public class ch18ex5 extends Example {
    public function ch18ex5() {
      //import TLF markup
      XML.ignoreWhitespace = false;
      var tlfxml:XML =
<TextFlow xmlns="http://ns.adobe.com/textLayout/2008">
<format name="red" id="red" color="0xff0000"/>
<p fontFamily="Calibri, Verdana, _sans" fontSize="28">Mind your
<span format="red">p</span>s and <span format="red">q</span>s!</p>
</TextFlow>;
      var textFlow:TextFlow =
        TextConverter.importToFlow(tlfxml, TextConverter.TEXT_LAYOUT_FORMAT);

      //display TextFlow
      var factory:TextFlowTextLineFactory = new TextFlowTextLineFactory();
      factory.compositionBounds = new Rectangle(0, 0, 500, 500);
      factory.createTextLines(this.addChild, textFlow);

      //export as TextField markup
      var tfhtml:String = TextConverter.export(textFlow,
        TextConverter.HTML_FORMAT, ConversionType.STRING_TYPE) as String;
      trace("\n\n\n\n\n\n\n" + tfhtml);

      //display TextField
      var tf:TextField = new TextField();
      tf.y = 50;
      tf.autoSize = TextFieldAutoSize.LEFT;
```

EXAMPLE 18-5 *(continued)*

```
      addChild(tf);
      tf.htmlText = tfhtml;
    }
  }
}
```

The example first imports TLF markup into a flow. Then the example exports the text using HTML formatting, finally displaying the converted text in a `TextField` to prove that the conversion worked. It's important to configure the XML parser not to ignore whitespace when writing text in XML. `TextConverter` supports these formats:

- `TextConverter.FXG_FORMAT` — FXG format, an XML-based graphics exchange format similar to SVG, that can be used in Flex 4, Flash Catalyst, and other applications.
- `TextConverter.HTML_FORMAT` — The subset of HTML supported by `TextField`. Covered in Chapter 17.
- `TextConverter.PLAIN_TEXT_FORMAT` — Plaintext, stripped of formatting.
- `TextConverter.TEXT_LAYOUT_FORMAT` — TLF markup. Only TLF format preserves all the intricate layout details of a `TextFlow`.

TLF markup is a portable and concise way to store content for the Text Layout Framework.

Available Formatting Options

There are dozens of formatting options available in the TLF. Some of these, like columns, are unique to the TLF, whereas many of them are provided directly by the FTE. Describing all of them would take more room than I have here, so see the AS3LR for details. The documentation on the `ITextLayoutFormat` interface explains all available formatting options. Find it at `http://bit.ly/ITextLayoutFormat`.

Editing Features

You can endow any text flow with the ability to select or edit text. Because the TLF is so modular, the classes that control text selection and editing are separate classes called *interaction managers*. The interaction managers plug into the mouse events dispatched by each container in a text flow and add the appropriate behaviors.

The interaction managers in the TLF are found in the `flashx.textLayout.edit` package. `SelectionManager` lets you select and copy text; `EditManager` extends `SelectionManager`, adding full text editing on top of selection. Again, the having a choice means you can make text behave to your specifications while minimizing dependencies on the framework and thus code size. Both of these interaction managers provide programmatic access in addition to user interactivity; for example, you can select portions of text without user intervention by calling `selectRange()` on the `SelectionManager` instance.

To bestow these abilities on a story, assign an instance of one of these classes (or your own, as the TLF is extensible) to the `interactionManager` property of the `TextFlow` instance. Of

course, for the interactivity to kick in, the story must be composed on-screen with a flow composer. Remember that `TextLine` factories just spit out `TextLines`, so they can't be interactive. It's the `ContainerControllers` that assist `SelectionManager` and `EditManager`.

You created a simple `SelectionManager` in Example 18-4. In Example 18-6, you'll put a lot of skills together: you'll load TLF markup externally that uses a combination of formatting, apply it to a series of connected containers, set up some properties of the containers you haven't seen until now, add text editing to the flow, and use ActionScript to set a link in the content without user intervention.

EXAMPLE 18-6 `http://actionscriptbible.com/ch18/ex6`

Editable Text in Linked Containers

```
package {
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.utils.setTimeout;
  import flashx.textLayout.compose.StandardFlowComposer;
  import flashx.textLayout.container.ContainerController;
  import flashx.textLayout.container.ScrollPolicy;
  import flashx.textLayout.conversion.ITextImporter;
  import flashx.textLayout.conversion.TextConverter;
  import flashx.textLayout.edit.EditManager;
  import flashx.textLayout.elements.TextFlow;
  import flashx.undo.UndoManager;
  public class ch18ex6 extends Sprite {
    protected var textFlow:TextFlow;
    protected var editManager:EditManager;
    public function ch18ex6() {
      var loader:URLLoader = new URLLoader(new URLRequest(
        "http://actionscriptbible.com/files/alice-ch2-tlf.xml"));
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onLoadComplete(event:Event):void {
      var tlfMarkup:String = URLLoader(event.target).data;
      var flowComposer:StandardFlowComposer = new StandardFlowComposer();
      var importer:ITextImporter =
        TextConverter.getImporter(TextConverter.TEXT_LAYOUT_FORMAT);
      importer.throwOnError = true;
      textFlow = importer.importToFlow(tlfMarkup);
      textFlow.flowComposer = flowComposer;

      var l:Loader = new Loader(); addChild(l);
      l.x = 10; l.y = 130;
      l.load(new URLRequest("http://actionscriptbible.com/files/alice-1.jpg"));

      var sprite:Sprite, container:ContainerController;
      sprite = new Sprite(); addChild(sprite);
```

EXAMPLE 18-6   *(continued)*

```
    sprite.x = 10; sprite.y = 10;
    container = new ContainerController(sprite, 480, 120);
    container.horizontalScrollPolicy = ScrollPolicy.OFF;
    container.verticalScrollPolicy = ScrollPolicy.OFF;
    flowComposer.addController(container);

    sprite = new Sprite(); addChild(sprite);
    sprite.x = 210; sprite.y = 130;
    container = new ContainerController(sprite, 280, 170)
    container.horizontalScrollPolicy = ScrollPolicy.OFF;
    container.verticalScrollPolicy = ScrollPolicy.OFF;
    flowComposer.addController(container);
    sprite = new Sprite(); addChild(sprite);

    sprite.x = 10; sprite.y = 310;
    container = new ContainerController(sprite, 480, 300)
    container.horizontalScrollPolicy = ScrollPolicy.OFF;
    container.verticalScrollPolicy = ScrollPolicy.OFF;
    flowComposer.addController(container);

    editManager = new EditManager(new UndoManager());
    textFlow.interactionManager = editManager;
    textFlow.flowComposer.updateAllControllers();

    setTimeout(demonstrateEdits, 2000);
  }
  private function demonstrateEdits():void {
    try {
      var pointer:* = textFlow.getChildAt(0); //textFlow>_div_
      pointer = pointer.getChildAt(1); //textFlow>div>paragraph1, _paragraph2_
      editManager.selectRange(pointer.getAbsoluteStart() + 1, 40);
      editManager.applyLink("http://actionscriptbible.com");
      textFlow.flowComposer.updateAllControllers();
    } catch(error:Error) {
      trace("Error editing: " + error.toString());
    }
  }
}
}
```

If you can, try running the example and editing the content of the story. As expected, editing the text in any container causes the whole layout to reflow. At all times, the containers prevent text from bleeding over the figure that's floated to the left with the Loader instance.

To get more experience with TLF markup, you can also look at the XML file that this example loads. It uses a few special styles for directions that Alice imagines writing, and for the title. You can try

modifying the tab stops to change how Alice's directions to her right foot look. This, and the song she sings later, uses tab stops to present the text in a more interesting way.

Finally, you might try pressing Command-Z or Control-Z after making an edit to the text. If all goes well, you should see your change revert, because you've also included undo support.

Undo and Redo History

The `EditManager` interaction manager allows you to attach an `UndoManager` as well, adding undo and redo support to any `TextFlow`. This is another example of the TLF's modularity. Each class or set of classes adds on just one behavior or one cohesive set of behaviors. The `UndoManager`, found in the `flashx.undo` package, works with the `EditManager`, recording the user's actions editing the associated `TextFlow`. It keeps two stacks of edit operations so that you can not only undo a change, but redo it as well. The first stack is your usual history stack, but as you pop an edit operation off the history stack, it is added to the redo stack so that it can be retained if you decide you want to do that edit after all.

Because the `UndoManager` needs to work directly with the `EditManager`, the `EditManager` owns it. You assign it to the `EditManager` using its constructor, as shown in Example 18-6. As you saw in the example, no further action is required; now undo and redo are fully supported just as you'd expect.

Programmatic Editing

All the managers that imbue a `TextFlow` with interactivity just snap in and work. But if you care to dive in, you'll notice that the managers are transparent. You can reach in and perform all the same actions using ActionScript as you would using your mouse and keyboard.

In fact, it's this feature that makes undo and redo so easy. As in many applications, undo and redo are implemented with the *command pattern*: every action is represented by a command object, which can be executed, and in this case, undone. Edit operations are represented by command objects found in the `flashx.textLayout.operations` package, all of which implement `IOperation`. The interface defines `performUndo()` and `performRedo()` methods, enabling the `UndoManager` to perform its magic.

In practice, you'll call methods directly on the interaction managers instead of using commands. Table 18-2 explains some of the more useful methods you can use to perform selections, edits, and undo/redo. Remember that `EditManager` extends `SelectionManager`, so all methods that apply to `SelectionManager` apply to `EditManager` as well.

You'll notice that a selection can be defined by more than just a beginning and an end. It is fully described by a `SelectionState`, which contains the properties `absoluteStart`, `absoluteEnd`, `anchorPosition`, and `activePosition`. The absolute start and end refer to the character indices that the selection begins and ends at, as you'd imagine. But the anchor position and active position refer to the selection as the user created it; they encode the direction that the user selected text in. You can easily start a selection later in the text than you end it: simply hold down Shift while pressing the left arrow, or shift-click to the left of the cursor (in left-to-right text). The `activePosition` of a selection is the location of the cursor. Even if no text is selected, the `SelectionState` will indicate a zero-length selection at the position of the cursor, so you can retrieve the cursor location as follows:

```
//assume textFlow is a selectable text flow
var cursorPosition:int = textFlow.interactionManager.activePosition;
```

Clipboard

You can cut and paste text in a text flow by using the `cutTextScrap()` and `pasteTextScrap()` methods of the `EditManager` mentioned in Table 18-2. These return and take instances of the `TextScrap` class, which is basically opaque. It doesn't store a copy of the text inside it, and you can't access this text.

You can create a new `TextScrap` programmatically by using its `createTextScrap()` static method. Send the method a `TextRange` (the class `SelectionState` is based on) and it will create a `TextScrap` associated with that range of text.

TABLE 18-2

Selected Interaction Manager Methods

Class	Method or Property	Description
`SelectionManager`	`selectRange()`	Selects a range of characters from `anchorPosition` through `activePosition`, leaving the cursor at `activePosition`
	`selectAll()`	Select all text in the flow
	`absoluteStart,` `absoluteEnd,` `activePosition,` `anchorPosition`	The properties of the current selection
	`getSelectionState()`	Returns a `SelectionState` object defining the current selection
	`editingMode`	How the associated `TextFlow` may be interacted with: `EditingMode.READ_ONLY`, `EditingMode.READ_SELECT`, or `EditingMode.READ_WRITE`.
	`focused`	Whether the text flow has focus
	`setFocus()`	Give focus to the text flow
`EditManager`	`EditManager.overwriteMode`	(static) Toggles between insert and overwrite keyboard modes
	`undo()`, `redo()`	Undo or redo an operation; see also methods of `UndoManager`
	`beginCompositeOperation(),` `endCompositeOperation()`	Group the operations between these calls into a composite operation, which can be reverted with a single undo command
	`doOperation()`	Execute an operation on the text flow

Class	Method or Property	Description
	`applyFormat()`	Apply container, paragraph, and character formats to applicable elements in the selection
	`applyContainerFormat()`, `applyParagraphFormat()`, `applyLeafFormat()`	Apply the appropriate kind of format to applicable elements in the selection
	`applyLink()`, `applyTCY()`	Change selected text into a link element or TCY element, or back into a span
	`insertText()`, `insertInlineGraphic()`	Insert text or inline graphics at the active selection position
	`deleteText()`	Delete the selected text
	`deleteNextCharacter()`, `deletePreviousCharacter()`, `deleteNextWord()`, `deletePreviousWord()`	Delete the character or word neighboring the active selection position
	`splitParagraph()`	Split the paragraph at the active selection position, creating a new `ParagraphElement`
	`cutTextScrap()`	Delete the selection, returning it as a `TextScrap`
	`pasteTextScrap()`	Insert the `TextScrap`, replacing the current selection
`UndoManager`	`undo()`, `redo()`	Undo or redo the most recent operation
	`canUndo()`, `canRedo()`	Return whether you can perform an undo or redo
	`peekUndo()`, `peekRedo()`	Return the next undo or redo operation without removing it from the stack
	`popUndo()`, `popRedo()`	Return the next undo or redo operation as an `IOperation`, removing it from the stack but without performing it
	`pushUndo()`, `pushRedo()`	Adds an operation to the undo or redo stack

Using `TextScraps`, you can set and load the contents of the system clipboard with
`TextClipboard`'s `setContents()` and `getContents()` methods. The Flash Player security
model prevents unfettered access to the clipboard, however. You're allowed to read the clipboard if
the user has chosen to paste text into a text flow, and you're allowed to set text to the clipboard in
response to any user input.

In Example 18-7, you'll use user input to copy a text flow to the clipboard manually, without the text
being editable.

EXAMPLE 18-7   `http://actionscriptbible.com/ch18/ex7`

Copying to the Clipboard

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.text.*;
  import flashx.textLayout.compose.StandardFlowComposer;
  import flashx.textLayout.container.ContainerController;
  import flashx.textLayout.conversion.TextConverter;
  import flashx.textLayout.edit.TextClipboard;
  import flashx.textLayout.edit.TextScrap;
  import flashx.textLayout.elements.*;
  public class ch18ex7 extends Sprite {
    protected var flow:TextFlow;
    public function ch18ex7() {
      flow = TextConverter.importToFlow("Cellar door",
        TextConverter.PLAIN_TEXT_FORMAT);
      flow.fontFamily = "_typewriter";
      flow.fontSize = 11;
      flow.flowComposer = new StandardFlowComposer();

      var container:Sprite = new Sprite();
      addChild(container); container.x = 10; container.y = 10;
      var containerController:ContainerController =
        new ContainerController(container, 300, 20);
      flow.flowComposer.addController(containerController);
      flow.flowComposer.updateAllControllers();

      var button:TestButton = new TestButton(100, 20, "Copy");
      button.addEventListener(MouseEvent.CLICK, onClick);
      addChild(button); button.x = 320; button.y = 10;
```

```
        //textfield to try pasting to afterward
        var tf:TextField = new TextField();
        tf.type = TextFieldType.INPUT;
        tf.defaultTextFormat = new TextFormat("_typewriter", 11);
        tf.width = 410; tf.height = 100;
        tf.border = true;
        tf.wordWrap = tf.multiline = true;
        tf.text = "Try pasting the result here.";
        addChild(tf); tf.x = 10; tf.y = 40;
    }
    protected function onClick(event:MouseEvent):void {
      TextClipboard.setContents(
        TextScrap.createTextScrap(new TextRange(flow, 0, flow.textLength)));
    }
  }
}
import flash.display.*;
import flash.text.*;
class TestButton extends Sprite {
  public var label:TextField;
  public function TestButton(w:Number, h:Number, labelText:String) {
    graphics.lineStyle(0.5, 0, 0, true); graphics.beginFill(0xa0a0a0);
    graphics.drawRoundRect(0, 0, w, h, 8);
    label = new TextField(); addChild(label);
    label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
      false, null, null, "center");
    label.width = w; label.height = h; label.text = labelText;
    label.y = (h - label.textHeight)/2 - 2;
    buttonMode = true; mouseChildren = false;
  }
}
```

The highlighted line, triggered on a mouse click, creates the text scrap and sets it to the clipboard. A traditional `TextField` is provided to test the contents of your clipboard and convince yourself it's working.

Events

Finally, no interactive text manager would be complete without its fair share of events. All the pertinent events in the TLF are dispatched by the `TextFlow`. (You may have noticed by now that the model, `TextFlow`, is far more important than any controller in the TLF architecture; it stores references to all the associated helper classes and dispatches the events.) These are summarized in Table 18-3.

TABLE 18-3

Selected TextFlow Events

Event name	Description
`FlowElementMouseEvent.CLICK,` `MOUSE_DOWN, MOUSE_UP,` `ROLL_OVER, ROLL_OUT`	Mouse action occurred on a link in the text flow. To click on a link in editable text, you must hold down the Control or Command key while clicking. Is also dispatched by individual `LinkElements`.
`CompositionCompleteEvent` `.COMPOSITION_COMPLETE`	The flow has finished being (re)composed.
`StatusChangeEvent` `.INLINE_GRAPHIC_STATUS_CHANGE`	An inline graphic changed size or finished loading. Its new status appears in the `status` property of the event object.
`SelectionEvent` `.SELECTION_CHANGE`	The text flow's selection was modified. The `selectionState` property of the event object stores the new selection.
`FlowOperationEvent` `.FLOW_OPERATION_BEGIN,` `FLOW_OPERATION_END`	An edit operation began or finished. The event object's `operation` parameter stores the operation. Call `preventDefault()` on the event object to cancel the operation.
`TextLayoutEvent.SCROLL`	A text container was scrolled.

Flow and Container Configuration

You can set options on any text container controller that fine-tune its detail and appearance. You do so with the `Configuration` class, which can be assigned to a flow by passing it to the `TextFlow` constructor, or to the `TextConverter` when importing text. These options fall under a few major categories:

- Event handling — The `manageEnterKey` and `manageTabKey` properties control whether the events for these keys are handled internally by the TLF, or available for your code to handle. Turn on `manageTabKey`, for instance, and you can get the Tab key to indent text in a TLF story, rather than advance focus through the tabbing order.

- Scrolling — Properties in the `Configuration` let you set how the text scrolls due to selection, how much the mouse wheel scrolls, or how much a Page Down/Page Up key will scroll text.

- Selection formats — Using the `SelectionFormat` object, you can define how the cursor and selection are drawn, including the color and alpha of the selection and the caret, and even the blink rate of the caret. Separate properties of `Configuration` let you set different selection styles depending on whether the window or the text flow have focus.

- Link formats — If not set individually with the `linkNormalFormat`, `linkHoverFormat`, and `linkActiveFormat` properties of a `FlowElement`, or set via corresponding tags in TLF markup, you can set defaults that apply to all text in the story by using these same properties on the `Configuration` object.

A TextField Adapter Class

To truly blow your mind, the TLF comes with a UI control that implements the same interface as `TextField`, but using the Text Layout Framework to lay out its content. The `TLFTextField`,

found in the `flashx.textLayout.controls` package, has the methods and properties of a `TextField`. However, some are there for show only: not all the features of `TextField` are duplicated. The most notable exceptions are editing, selection, scrolling, and text measurement. Okay, so it's not perfect, but it has its place.

The `TLFTextField` control is useful to update legacy `TextField` code to use when you're merging legacy code and the TLF, especially if you need to share a font that's embedded using the newer DefineFont4, because the original `TextField` won't use those fonts.

Fonts Revisited

Along with the new Flash Text Engine, Flash Player 10 adds a new format for font embedding and a new font rasterizer. The DefineFont4 rasterizer and CFF font embedding are used in all Flash Text Engine and Text Layout Framework applications. Flash Player 10 also greatly improves the quality of device text, using advanced font rendering available on the host OS. Because this builds on existing technologies, and Flash Player is backward-compatible, old text renderers are still available. Table 18-4 summarizes the kinds of text rasterization used in Flash Player 10 and later.

The most important thing to realize is that there is a relationship between the text display components, the kind of font embedding you use, and the font rasterizer used. You have to consider all three. The new Flash Text Engine only works with device fonts and fonts embedded with DefineFont4; it can use the new device font rasterizer, the classic Flash outline rasterizer, or the CFF rasterizer. Conversely, the old `TextField` works only with device fonts and fonts embedded with DefineFont3; it can use the new device font rasterizer, the classic Flash outline rasterizer, or the Saffron type engine.

TABLE 18-4

Flash Player Text Rasterization

Rasterizer	Available in	How to Use	Limitations and Notes
Device Fonts	< FP10	Reference an installed font that is not embedded	Cannot be rotated; aliased
Flash Outline Rasterizer	All	Embed with DF3 and use `AntiAliasType.NORMAL`; or embed with DF4 and use `RenderingMode.NORMAL`	Anti-aliased but not crisp
Saffron	FP8	Embed with DF3, use `AntiAliasType.ADVANCED`	Works with `TextField` and Halo Flex components; looks great
Device Fonts (new)	FP10	Reference an installed font that is not embedded	Appearance controlled by OS; can't guarantee font is installed
CFF Rasterizer	FP10	Embed with DF4/CFF; use `RenderingMode.CFF`	Works with FTE, TLF, and Spark Flex components; looks great

Both the old Saffron rasterizer and the new CFF rasterizer produce excellent results. To compare these, check out a demonstration from the Adobe Typography Blog at `http://bit.ly/df3-vs-df4`.

Embedding CFF Fonts
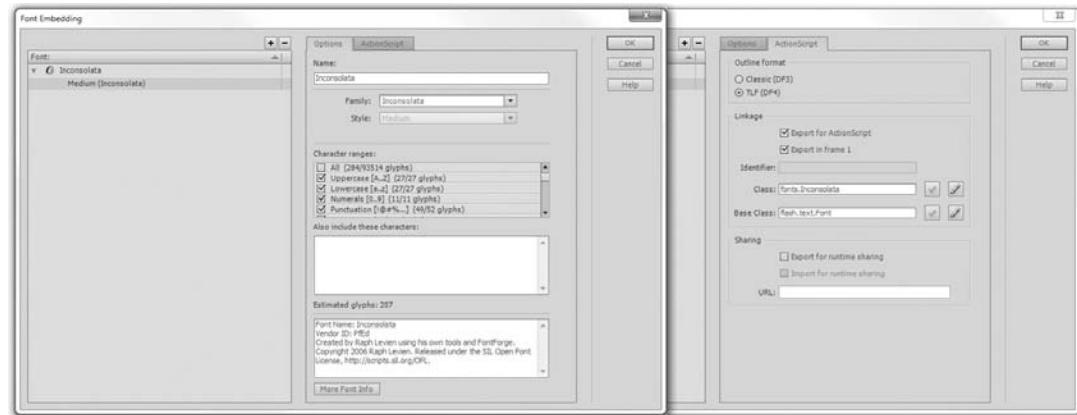
If you're going to use the new Flash Text Engine in any capacity, you have to embed your fonts in the Compact Font Format (CFF), using DefineFont4. Thankfully, as long as you know which font embedding format to use, the methods don't differ much. Again, you can use either Flash Professional or the Flex SDK (including any IDE that uses it to compile, such as Flash Builder) to embed your fonts.

Using Flash Professional

Flash Professional CS5 adds a new Font Embedding panel that makes embedding a font using DefineFont3 or DefineFont4 easy. To access it, create a new Font symbol in the Library. You'll see the panel, pictured in Figure 18-5.

FIGURE 18-5

Flash Professional's font embedding panel

In the Options tab, choose the font families and faces you'd like to embed, and give the font a name. The name you choose for it in this panel will be the definitive name of the font to Flash Player at runtime. You'll retrieve the font later by referencing this name in the `fontFamily` property of an `ITextLayoutFormat`; its original name is immaterial. In Figure 18-5, I've named the font Inconsolata by its original name, which is acceptable and certainly sensible.

In the ActionScript tab, once you've chosen to export the font for ActionScript, you have the option to select whether it's embedded using DefineFont3 or DefineFont4. Check TLF (DF4) to enable the font for use in the FTE, TLF, or Flex Spark components.

See Chapter 17 for a refresher on the other font embedding parameters, if you need it.

Using the Embed Tag

If you're using the Flex SDK or a Flex SDK-based IDE, you can use the `[Embed]` metadata tag to include a font in the compiled SWF. You saw how this worked for DefineFont3 fonts in Chapter 17.

To embed a font as CFF (using DefineFont4), simply add the attribute

```
embedAsCFF="true"
```

to any font embedding metadata tag. You can set the ranges of characters to be included using Unicode, as before. Setting up named Unicode ranges can make this process more palatable: find out how at `http://bit.ly/embed-named-unicode-ranges`.

```
[Embed(
  source="/../etc/Inconsolata.otf",
  fontFamily="Inconsolata",
  embedAsCFF="true",
  unicodeRange="U+0020-U+002F,U+0030-U+0039,U+003A-U+0040,\
                U+0041-U+005A,U+005B-U+0060,U+0061-U+007A,U+007B-U+007E"
)]
private var InconsolataEmbed:Class;
```

The preceding snippet embeds the OpenType Inconsolata font using the same font name, in CFF, and limited to basic Latin characters.

Font Lookup

You can still use the `Font` class to list available fonts and examine the properties of these fonts; fonts are still represented at runtime as subclasses of `Font`. Fonts that are embedded using DefineFont4 have a `fontType` property of `FontType.EMBEDDED_CFF`; you can use this property to identify them.

More importantly, the Flash Text Engine introduces a new way to look for and describe fonts that I mentioned way back at the beginning of this chapter, called `FontDescription`. The same properties of `FontDescription` become some of the many properties of `ITextLayoutFormat` in the TLF. In either case, the combination of these properties determines how Flash Player decides what font to draw.

Both interfaces include these properties:

- `fontFamily` — The name of the font set when embedding it
- `fontPosture` — Whether the font is italic (`FontPosture.ITALIC`) or not (`FontPosture.NORMAL`)
- `fontWeight` — Whether the font is bold (`FontWeight.BOLD`) or not (`FontWeight.NORMAL`)
- `fontLookup` — Where to look for a font with this name, weight, and posture: either in system fonts (`FontLookup.DEVICE`) or in embedded fonts (`FontLookup.EMBEDDED_CFF`)
- `renderingMode` — When rasterizing embedded text, whether to use the CFF rasterizer (`RenderingMode.CFF`) or the classic Flash outline rasterizer (`RenderingMode.NORMAL`). See Table 18-4.

397

- `cffHinting` — When using the CFF rasterizer with DefineFont4 embedded fonts, defines whether to optimize anti-aliasing for small font sizes, snapping horizontal stems that might otherwise be lost in blending to the subpixel grid (`CFFHinting.HORIZONTAL_STEM`) or to anti-alias mathematically, with no hints (`CFFHinting.NONE`).

These properties present a different way of requesting certain font features from the text engine than you learned for `TextField`. Remember that the `fontFamily` property, specifically, can take a comma-separated string so that a preferred series of fallback fonts may be specified. If you're trying to visualize some of the more advanced features, you can see the effect of the `renderingMode` and `cffHinting` flags in the aforementioned demonstration at `http://bit.ly/df3-vs-df4`.

Caution

Forgetting to change the font lookup path to embedded fonts is a common error. Because you presumably had the font installed to work with it in the first place, you might see the font as intended; but really you're looking at the device font. Test your application on computers that don't have the fonts installed, and make sure to set the `fontLookup` property of text flows when they use embedded fonts. ■

Summary

- The Flash Text Engine (FTE) enables much richer text in Flash Player.
- `TextField` still exists and still has its purposes.
- The Text Layout Framework (TLF) extends the FTE to provide even more functionality.
- The TLF is written in pure ActionScript 3.0 and thus can be modified and extended.
- The TLF uses a hierarchy of content elements rooted in a single `TextFlow`.
- TLF uses either a `TextLine` factory or a flow composer to lay out its text.
- TLF markup mirrors class relationships and properties that can be set in code.
- TLF markup is imported and exported with `TextConverter`.
- TLF text flows may be selected or edited with the appropriate interaction manager.
- To use TLF or FTE with embedded fonts, you must embed the font as CFF.

Printing

I n this chapter, you learn how to print Flash content using the `PrintJob` class. You'll control what and how to print and even print documents differently than they appear on-screen. You'll also look at different approaches to printing.

Why Print from Flash?

ActionScript is used in many different contexts. For some of the more "traditional" applications, like a Flash banner, printing is out of the question. But with ActionScript being deployed in enterprise-level Flex applications and on the desktop in AIR applications, plenty of situations demand the ability to print, like a timesheet application or ticketing software for an airline.

Additionally, when you're talking about browser-based applications, browsers might not print the contents of a plug-in like Flash Player, so you may have no choice but to develop your own printing scheme.

Even better, when you use the print facilities in ActionScript, you have a lot of control over how things print. You can print multipage documents and control where they break pages; you can be aware of the printer's properties and use the correct page size, orientation, and margins; and you can create your own headers and footers. Of course, printing from Flash Player gives you the advantage of its display capabilities, such as precise layout, high-quality and low-file-size vector graphics, embedded custom fonts, and image processing using filters and `BitmapData`, to name a few. Many of these capabilities are not available when using a browser to print.

Although printing with Flash Player gives you, the developer, a lot of control, you may want to give some of that control back to your users. If you don't need any of those advantages and would rather the users be presented with their familiar browser print experience, you can take another approach, which I'll just mention here. One option is to open a new browser window with the content to be printed populated by a server-side script or JavaScript, using `ExternalInterface`. Another option is to use a PDF generation library, such

as AlivePDF(`http://alivepdf.bytearray.org/`), which lets the user not only print the document but save and share it easily, or even put it on e-reader devices. If the output of your application is destined for the internet, these may also be desirable alternatives, because HTML and PDF are generally quite accessible and can be indexed and searched easily. If the document is just destined for the printer and you require control over how the document prints, it's best to go with a `PrintJob`.

Controlling Printer Output from Flash

With the `flash.printing.PrintJob` class, you can define how pages are constructed and sent to the printer. This section describes each of the methods and properties of the `PrintJob` class and explains how each works. If you want to see the `PrintJob` API in action, continue to the section "Adding Print Functionality to Applications."

Introducing the PrintJob Class

On the surface, there isn't too much to `PrintJob`. In fact, the class has only three methods. To create a new instance of the `PrintJob` class, use the constructor:

```
var printJob:PrintJob = new PrintJob();
```

The constructor takes no arguments. Once you have a `PrintJob` object, you initiate the three methods of the object, in the following order:

- `start()` — Opens the Print dialog box on the user's operating system. If the user clicks the Print (or OK) button in the Print dialog box, the method returns a `true` value. If the user cancels the dialog box, the method returns a `false` value. You should use the other two methods only if the `start()` method returns `true`.

- `addPage()` — Called sequentially, adds pages to the `PrintJob` object and tells it what to print from the display list. This method uses a number of arguments, which are discussed in the following sections.

- `send()` — Finalizes the output and sends the data to the printer's spooler.

Once you have sent the output to the printer with the `send()` method, you can delete the `PrintJob` object. Let's take a closer look at the `start()` and `addPage()` methods.

Starting a Print Request

When you call the `start()` method, Flash opens a new Print dialog box that prompts the user to accept or cancel the print request. It also allows the user to determine essential properties of the print job — such as the printer to use, the page size, and the page orientation — and gives Flash Player access to these choices.

The following properties are set on the `PrintJob` instance if the user clicks OK to a Print dialog box initiated from a Flash application. Some of the properties are measured in *points,* abbreviated as *pt.* There are 72 points to one inch. It's important to remember that points, millimeters, and inches are physical measurements, unlike pixels, which can appear at different physical sizes depending on the size and resolution of the display. Also, all the properties are of type `int`, so they are accurate to 1pt, or 1/72″.

- `paperHeight` — Represents the height (in points) of the paper size that the user has selected. For example, if the user has selected a paper size of 8.5″ × 11″, `paperHeight` returns a value of 792 points (11 in × 72 pt/inch = 792 pt).

- `paperWidth` — Represents the width (in points) of the paper size that the user has selected. 8.5″ × 11″ paper has a `paperWidth` value of 612 points.

- `pageHeight` — Perhaps the more useful of the height-based properties, the `pageHeight` property returns the height (in points) of the actual printable area. Many printers can print only to a certain portion of the paper size, leaving a margin around the edges of the paper. For example, on an 8.5″ × 11″ piece of paper, a typical printer can print an area sized 8.17″ × 10.67″. If you are trying to size output to the page, you should use this property over `paperHeight`, or your document might appear cropped.

- `pageWidth` — Like `pageHeight`, this represents the width (in points) of the actual printable area on the paper.

- `orientation` — Contains a string value of either `PrintJobOrientation.PORTRAIT` or `PrintJobOrientation.LANDSCAPE`, based on the user's setting in the Print dialog box. Landscape pages are longer than they are high, and portrait pages are taller than they are wide. This will be reflected in the `width` and `height` properties of `PrintJob`.

The `start()` method is synchronous. That means that it effectively pauses Flash until the user clicks the OK or Cancel button in the Print dialog box.

Note

Users must have a print driver installed to print from Flash or any other application. If they don't have a printer, they can still install a print driver to print to a file, such as the open source PDFCreator for Windows (`http://pdfforge.org/`). ∎

Determining the Print Target and its Formatting Options

So far you've only seen how to start and view the properties of a print job. To put content in that print job, you need to become familiar with the `addPage()` method. The `addPage()` method uses the following syntax, where `printJob` represents a `PrintJob` instance:

```
printJob.addPage(target, printArea, printOptions, frame);
```

The parameters are as follows:

- `target` — The `Sprite` that you want to print.

- `printArea` — A `Rectangle` instance whose properties determine the margins of the printable target. This parameter is optional; if it is omitted, the entire area of the target sprite is printed.

Note

The print area's coordinates are represented in the local coordinate system of the target sprite you are printing. ∎

- `printOptions` — A `PrintJobOptions` instance that determines how the target's contents are sent to the printer. By default, all contents are sent as vector artwork. However, if you specify a `PrintJobOptions` value, you can control whether or not the page is printed as a bitmap. The `PrintJobOptions` constructor accepts a Boolean parameter. If the property is set

to `true`, the artwork is rendered as a bitmap and then sent to the printer. If the property is set to `false`, the artwork is rendered in vectors and then sent to the printer. See the sections "Printing targets as vectors" and "Printing targets as bitmaps" for more information.

● `frame` — The frame number of the target movie clip to print. (Because `MovieClip` is a subclass of `Sprite`, you can specify a `MovieClip` instance as the `target` parameter.) If you want to print a specific frame of the target, you can use this optional parameter. If you omit this parameter, the current frame of the target is printed. Note that any ActionScript code on the specified frame will not be executed. As such, this is more useful for printing static content. In most cases, you will customize the print contents in a live display list and then print as-is.

You apply these parameters in later examples of this chapter. In the next sections, you learn more specifics of the `addPage()` parameters and how they affect the printed output from the Flash application.

Printing Targets as Vectors

You should omit the `options` parameter or use a `PrintJobOptions` instance whose `printAsBitmap` parameter is set to `false` when you are printing vector graphics such as:

● Text contained within Static, Dynamic, or Input text fields

● Vector artwork created with Flash Professional or another vector authoring tool

● Non-`Bitmap` display objects without alpha, brightness, tint, advanced color effects, or filters applied, and without `cacheAsBitmap` set

Especially if your print content includes text, you will really want to print the content as vectors. Printers have a much higher resolution than the screen: typically around 300 or 600 dots per inch as opposed to screens, which can be as low as 72 pixels per inch. Because of this, what looks crisp on a screen usually looks disappointing when scaled up to fill the page. When you use vector printing, Flash will scale up the print content to fill the print area using lossless vectors that preserve all their sharp edges at the destination resolution.

Caution

Any alpha or color settings for symbol instances or artwork are ignored when printing as vectors. Bitmap images also print with more aliasing (that is, rough, pixelated edges) if they're printed as vector artwork. Printing as vector artwork also fills alpha channels of any bitmap images with solid white. ■

Printing Targets as Bitmaps

The `PrintJobOptions` parameter should be constructed with a value of `true`, or bitmap mode, when you are using a variety of sources for your artwork and content, such as:

● Display objects using alpha, brightness, tint, advanced color effects, or filters.

● Display objects containing imported or loaded bitmap images or programmatically created bitmaps. Although bitmap images can be printed as vector artwork, they appear sharper when printed as bitmaps. More important, bitmap images with alpha channels print correctly if the transparent areas of the alpha channel overlap other artwork.

What happens to vector artwork (including text) that is printed as bitmaps? This setting still prints vector artwork, but it won't be as crisp as artwork output with the vector setting. However, you might find the differences between bitmap and vector settings with vector artwork negligible — if you're ever in doubt, test your specific artwork with both settings and compare the output. The bitmap setting is usually the safest bet if you are using bitmap images and any alpha or color effects.

Scaling Screen Dimensions to Print Dimensions

Perhaps the most difficult concept to grasp with the addPage() method is how the target is sized to the printed page. Using a conversion formula, you can determine how large your target will print on the printer's paper:

$$1\,\text{pixel} = 1\,\text{point} = 1/72\,\text{inch}$$

Therefore, if you have a sprite containing a 400 × 400–pixel square, that artwork will print at roughly 5.5″ × 5.5″ on the printed page. Use this formula along with the scaleX and scaleY properties of the target Sprite to size the content to its desired size on the page. You can revert these properties after printing is done.

Potential Issues with the Flash-Printed Output

Watch out for the following two pitfalls with the addPage() method parameters, which can cause unpredictable or undesirable output from a printer:

- Device fonts — If at all possible, avoid using device fonts with the printed output. Make sure all text is embedded for each text field used for printable content. Text that uses device fonts will print — however, if you have several elements in addition to device font text, the device text may not properly align with other elements on the page.

- Background colors — If you are using a dark background color, make sure you account for how that will affect the printing. If necessary, you can add a filled rectangle behind your printable content inside the target sprite to increase visibility. For example, if you want to print black text on white, you can temporarily add a white rectangle behind the text within the target sprite as you send it to the printer.

Be sure to check your applications for these problems before you test your printed output from a Flash application.

Adding Print Functionality to Applications

In Example 19-1, you'll load text from the internet, display it on-screen, and print it. At first, using no options, it will print on one page, cutting off much of the text.

EXAMPLE 19-1 `http://actionscriptbible.com/ch19/ex1`

Printing without Options

```
package {
  import com.actionscriptbible.Example;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.printing.PrintJob;
  import flash.text.TextField;
  import flash.text.TextFieldAutoSize;
  public class ch19ex1 extends Example {
```

```
    private var printableContent:Sprite;
    private var textField:TextField;
    private var loader:URLLoader;

    public function ch19ex1() {
      //Load the text from a text file.
      loader = new URLLoader();
      loader.load(new URLRequest(
        "http://actionscriptbible.com/files/alice-ch1.txt"));
      loader.addEventListener(Event.COMPLETE, onLoadComplete);

      //Create a multiline text field that auto-sizes.
      textField = new TextField();
      textField.multiline = true;
      textField.wordWrap = true;
      textField.autoSize = TextFieldAutoSize.LEFT;

      //Create a sprite container for the text field,
      //and add the text field to it.
      printableContent = new Sprite();
      printableContent.addChild(textField);
    }

    private function onLoadComplete(event:Event):void {
      trace("Done. Printing...");
      textField.text = loader.data;
      var printJob:PrintJob = new PrintJob();
      if (!printJob.start()) {
        trace("Printing cancelled!");
        return;
      }
      printJob.addPage(printableContent);
      printJob.send();
      trace("Print job submitted!");
    }
  }
}
```

When you click OK in the Print dialog box, one page prints. That one page will be the first page of text. Example 19-2 shows a way to print the entire text, page by page.

Printing Page by Page

```
package {
  import com.actionscriptbible.Example;
  import flash.display.Sprite;
```

```
import flash.events.Event;
import flash.geom.Rectangle;
import flash.net.URLLoader;
import flash.net.URLRequest;
import flash.printing.PrintJob;
import flash.text.TextField;
import flash.text.TextFieldAutoSize;

public class ch19ex2 extends Example {
  private var printableContent:Sprite;
  private var textField:TextField;
  private var loader:URLLoader;

  public function ch19ex2() {
    trace("Loading text...");
    loader = new URLLoader();
    loader.load(new URLRequest(
      "http://actionscriptbible.com/files/alice-ch1.txt"));
    loader.addEventListener(Event.COMPLETE, onLoadComplete);

    //Create a multiline text field that auto-sizes.
    textField = new TextField();
    textField.multiline = true;
    textField.wordWrap = true;
    textField.autoSize = TextFieldAutoSize.LEFT;

    //Create a sprite container for the text field,
    //and add the text field to it.
    printableContent = new Sprite();
    printableContent.addChild(textField);
  }

  private function onLoadComplete(event:Event):void {
    trace("Done. Printing...");

    var printJob:PrintJob = new PrintJob();
    if (!printJob.start()) {
      trace("Printing cancelled!");
      return;
    }

    //size the text field to the page
    textField.height = printJob.pageHeight;
    textField.width = printJob.pageWidth;
    textField.text = loader.data;
    var pages:int = Math.ceil(textField.textHeight / printJob.pageHeight);

    //loop through each page
    for(var i:int = 0; i < pages; i++) {
      printJob.addPage(
        printableContent,
        new Rectangle(0, i * printJob.pageHeight,
```

continued

405

EXAMPLE 19-2 *(continued)*

```
        printJob.pageWidth, printJob.pageHeight)
      );
    }

    printJob.send();
    trace("Print job submitted!");
  }
 }
}
```

When you test printing this time, the Flash application prints as many pages as necessary to print the entire text. However, pages can be cut off between lines. As an exercise, see if you can modify the example using TextField's scroll properties to avoid this. Or you might want to combine the printing tasks you learned about in this chapter with the advanced text layout from Chapter 18.

Summary

- You can print many useful items from Flash movies, such as coupons, receipts, artwork, and product catalogs or datasheets.
- The PrintJob class has all the methods and properties necessary to print Flash content.
- The addPage() method of the PrintJob class enables you to control which sprite is printed and how it should be printed.
- You must scale your print content to fit the print area at 1 pixel per point.
- Printing as vectors is preferable when printing text to make it appear crisp.

Part IV

Event-Driven Programming

Events and the Event Flow

ActionScript 3.0 uses a powerful framework for handling internal communication known as the *event framework*. *Events* are messages that are sent between objects when an action, such as a button click, has taken place. This enables you to create functionality that occurs interactively and without the need for direct method calls to other classes.

The Flash Player API uses events for many purposes in a unified way. Events handle mouse actions, timers, networking, and asynchronous errors. In this chapter you'll learn about how these subsystems use the event framework.

I'll take a look at how `EventDispatcher` objects communicate and how the new `Event` objects are structured. I'll also check out some of the more advanced features of the event framework, including how to create your own custom events and how to use event bubbling.

Introducing Events

What exactly is an event? At its core, an event is an object that represents an occurrence and describes the conditions surrounding that occurrence. This includes (but is not limited to) a description of the event, called the *event type*, and the origin of the event, also known as the *event target*.

Events — and the Observer design pattern that the event framework follows — allow one object, called an *event dispatcher*, to trigger the actions of one or more other objects, called *event listeners*, without having to know anything about the structure of the other objects. Because the object broadcasting the event doesn't need to know what methods to call on the receiving objects, listeners can be added and taken away at any time without altering code or creating dependencies. That is what makes them so powerful.

To understand this better, let's look at a real-world example of how events work.
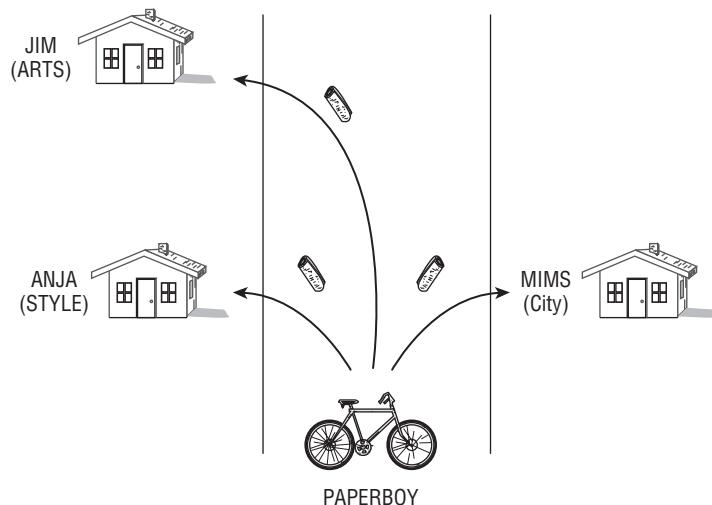
Saturday Morning Events

The relationship between an event dispatcher and an event listener is like that of a person who subscribes to a magazine or newspaper. Imagine it's a laid-back Saturday morning, your opportunity to wake up late, have a cup of coffee, and read the Saturday paper. Each Saturday, the paperboy delivers the paper to neighbors Mims, Anja, and Jim. Mims likes the City section, Anja loves Style, and Jim checks the Arts section for gallery openings. If you were to draw a diagram, it would look something like Figure 20-1.

FIGURE 20-1

The paperboy delivers the paper to all the subscribers. Everybody's happy!

This diagram shows almost the same thing that happens when an event is dispatched to its listeners. In fact, it's often said that a listener *subscribes to* an event dispatcher. Let's imagine this scene a little differently.

If you were describing this scenario with ActionScript, you might think of the paperboy as an event dispatcher, the newspaper subscribers as event listeners, and the newspaper as an event (or as a property of an event if you prefer). The action triggering the event in this case is the availability of a new paper. Like the newspaper example, each subscriber can choose how to respond to that event by reading a different section of the paper or by going to a particular reading spot.
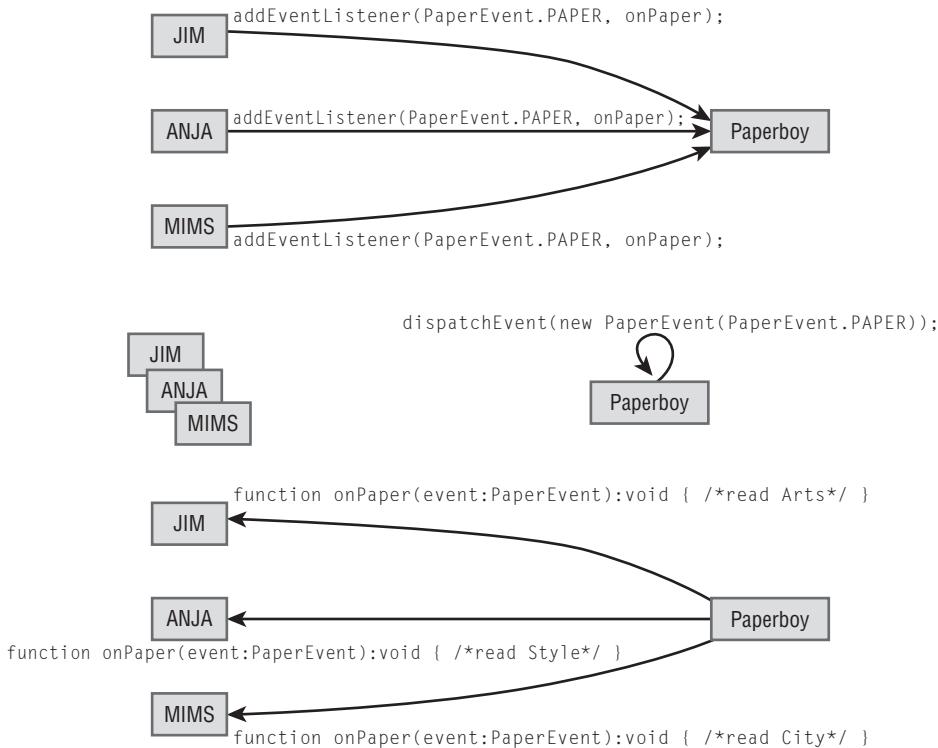
So, let's assume that the paperboy is an event dispatcher who fires off (or *dispatches*) an event of type `paperDelivered`. Each person uses the `addEventListener()` method to subscribe to the delivery event and sets up his version of the event handler `onPaperDelivered()`. When the `dispatchEvent()` method is called by the paperboy, every listener receives notice and is passed an `Event` object, as shown in Figure 20-2.

With a real newspaper subscription, if one of the subscribers decides she isn't interested in keeping up with the news anymore, she can easily unsubscribe by phoning the newspaper. The same thing works for event listeners. By using the `removeEventListener()` method, it's easy to unsubscribe from the events.

After you take a closer look at the different players and the terminology surrounding events, you'll be able to convert this pseudocode into real ActionScript.

The paperboy object sends a `paperDelivered` event to all the listeners.

Event Terminology

It will be easier to talk about event dispatchers once you have the terminology surrounding events down. The following sections define important terms and what they refer to or what roles they play.

Event

The term *event* could refer to a single occurrence, such as a mouse click or the end of a load sequence, or it could refer to the `Event` object that is dispatched during a `dispatchEvent()` call, such as an instance of the `MouseEvent` class.

Type

This is the type of event being broadcast, represented by a `String`. You might also think of this as being the title, kind, or name of the event. All calls to `dispatchEvent()` and `addEventListener()` must include a specific event type. The type of an event is how you

specify what kinds of events you're interested in being notified of when adding an event listener. Each `Event` class usually stores the names of its event types as static constants, and many `Event` classes have multiple types. For example, `MouseEvent.CLICK` is a string constant of `"click"` and is the event type of all mouse click events.

Target

When an event is broadcast, the *target* is the object that broadcasts the event, or the object that you're listening to. This can be a slightly confusing term if you expect the target to be the recipient of the event. You should think of targets from the event listener's point of view; they are a property of the `Event` object that refers back to the object of interest. A reference to the target object is always maintained during event propagation so that any event handler can know just who is dispatching the events.

Dispatcher

This term also refers to the object calling the `dispatchEvent()` method. *Dispatch*, *fire*, and *broadcast* are all terms variously used to describe the action of sending out an event. In other languages this might be called an *observable*, or a *notifier*.

Listener

A *listener* is an object that registers with a dispatcher to receive events. A listener might also be said to *listen for*, or *subscribe to*, events. A single dispatcher may have several listeners. In other languages this might be called an *observer*.

Handler

A function that is invoked when an event is received is called an *event handler*. It's also called a *listener function*.

Flow

The event system in ActionScript 3.0 works in a special mode for objects in a display list. When dealing with objects in the display list (such as sprites), some events are sent through from the root display object or stage, through every parent object of a given display object until reaching the target. After reaching the target, the event is sometimes sent back through every parent object until reaching the stage. This behavior is called the *event flow*. The event flow allows you to listen to events that occur on whole groups of display objects or to avoid manually passing on a message through intermediary objects that don't care about it. Objects not in the display list don't participate in the event flow. This may sound confusing, but I'll cover it in the section "The Event Flow."

Phase

In the event flow, the event's trip from the stage down to the dispatcher, the instant that it arrives at the dispatcher, and the trip back up to the stage are separate phases. They are known as the *capture phase*, *target phase*, and *bubble phase*, respectively. These, too, are covered in the section "The Event Flow."

Now that you have a better understanding of the terminology, let's dive into the classes of the event framework.

The EventDispatcher Class

At the root of event dispatching is the `EventDispatcher` class. You'll find that many classes in ActionScript 3.0 (including all display objects and many networking classes) already extend `EventDispatcher`, making the logistics of sending events quite convenient.

Using EventDispatcher

Let's walk through the process of triggering, dispatching, and receiving events. The first thing you need is an event dispatcher. In this example, let's create a `Thermometer` to measure temperatures. The `Thermometer` will also be able to notify anyone who's interested in the temperature. You'll do this by making `Thermometer` a subclass of `EventDispatcher`. You'll also add a public setter function for changing the temperature. In this setter, you'll call `dispatchEvent()`, defined in the superclass `EventDispatcher`, to notify other objects when the temperature changes:

```
import flash.events.*;
class Thermometer extends EventDispatcher {
  protected var _temp:Number = 72;
  public static const TEMP_CHANGED:String = "tempChanged";

  public function set temp(newTemp:Number):void {
    _temp = newTemp;
    dispatchEvent(new Event(TEMP_CHANGED));
  }

  public function get temp():Number {
    return _temp;
  }
}
```

Let's walk through some of the key lines more carefully.

```
protected var _temp:Number = 72;
```

`_temp` stores the temperature value. The default is 72 (since I'm in the United States where we use crazy units of measurement, this property is in degrees Fahrenheit).

```
public static const TEMP_CHANGED:String = "tempChanged";
```

`TEMP_CHANGED` is a simple string that you're going to use to define the type of event you're broadcasting. This string can be anything just as long as your event listeners are listening for the same string. The easiest way to do this is to store the string in a static constant; that way, it's always publicly available and can't be changed.

Note
You will often see event types defined as static constants of specific `Event` subclasses such as `MouseEvent.CLICK`. This is a good practice to use, but to keep things simple, let's define the event type in the `Thermometer` class. ■

```
                    dispatchEvent(new Event(TEMP_CHANGED));
```

This is where your event is being fired out to the listeners. The `dispatchEvent()` method takes a single argument, which is an `Event` object containing information about the event. In this case, you're creating a new `Event` with type `TEMP_CHANGED` and passing it directly to the `dispatchEvent()` call.

This class is pretty straightforward. Whenever the temperature changes, an event is fired.

Example 20-1 creates some listeners for this dispatcher.

EXAMPLE 20-1 http://actionscriptbible.com/ch20/ex1

Dispatching and Listening to Events

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  public class ch20ex1 extends Example {
    public function ch20ex1() {
      var t:Thermometer = new Thermometer();
      t.addEventListener(Thermometer.TEMP_CHANGED, onTempChanged);
      t.debugSimulateCrazyWeather();
    }
    protected function onTempChanged(event:Event):void {
      var t:Thermometer = Thermometer(event.target);
      trace("It's now " + t.temp.toFixed(1) + "°F");
    }
  }
}

import flash.events.*;
import flash.utils.setInterval;
class Thermometer extends EventDispatcher {
  protected var _temp:Number = 72;
  public static const TEMP_CHANGED:String = "tempChanged";

  public function set temp(newTemp:Number):void {
    _temp = newTemp;
    dispatchEvent(new Event(TEMP_CHANGED));
  }
  public function get temp():Number {
    return _temp;
  }
  internal function debugSimulateCrazyWeather():void {
    setInterval(function():void {temp += 3 * (Math.random() - 0.5);}, 1000);
  }
}
```

Let's step through the main example class now.

```
        t.addEventListener(Thermometer.TEMP_CHANGED, onTempChanged);
```

Here you tell the `Thermometer` instance to invoke the `onTempChanged()` function when the `TEMP_CHANGED` event is dispatched.

Note

In ActionScript 3.0, methods are *bound*; they remember the object they're associated with and execute in the correct scope independent of where they are called. That means that you don't have to pass a context, delegate function, or reference to `this` to an event dispatcher, only the function that you want to use as the callback. ■

```
public function onTempChanged(event:Event):void {
```

You define `onTempChanged` as a function that takes one argument: an `Event` object. You'll use this function to gather data about the event and then act on it.

In every case, event handlers should have exactly one argument: for the `Event` object. Sometimes you will actually use this event object and other times you won't. If you know that the event object won't be used, you might find it helpful to set the argument to `null` by default:

```
function onEvent(event:Event = null):void { ... }
```

This way, you will be able to call the function with or without an event object, making it more versatile. Back to the example:

```
var t:Thermometer = Thermometer(event.target);
```

By using the `event.target` property, you can get the original sender of the event, in our case, the `Thermometer`. The `target` property is of type `Object`, so you have to cast it to type `Thermometer` to access its `temp` property.

```
trace("It's now " + t.temp.toFixed(1) + "°F");
```

Finally, you trace out the value of the thermometer's temperature, only displaying one decimal place.

Running the code causes the `Thermometer` to change its own value every second, firing an event, which triggers the event handler, whose code runs and traces out the temperature. Not impressed?

I can't say I blame you. Why do you need to pass an event just to display the temperature? Well, this is a simplified setup for demonstration purposes, but as you'll see, events can become quite powerful.

Say you want to add another view that acts as a heat gauge. It should go from green to red as the temperature reaches an uncomfortable range, shown in Example 20-2. This code has little to do with the original class that simply traces out the temperature, but it can get information from the same source by adding an event listener.

EXAMPLE 20-2 `http://actionscriptbible.com/ch20/ex2`

Adding Another Event Listener

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
```

EXAMPLE 20-2   *(continued)*

```
public class ch20ex2 extends Example {
  public function ch20ex2() {
    var t:Thermometer = new Thermometer();
    t.addEventListener(Thermometer.TEMP_CHANGED, onTempChanged);

    var warn:TempWarning = new TempWarning(t);
    warn.x = stage.stageWidth - warn.width;
    addChild(warn);

    t.debugSimulateCrazyWeather();
  }
  protected function onTempChanged(event:Event):void {
    var t:Thermometer = Thermometer(event.target);
    trace("It's now " + t.temp.toFixed(1) + "°F");
  }
}
}
import flash.display.Sprite;
import flash.geom.ColorTransform;
class TempWarning extends Sprite {
  public function TempWarning(t:Thermometer) {
    graphics.beginFill(0xffff00);
    graphics.drawRect(0, 0, 50, 50);
    graphics.endFill();
    t.addEventListener(Thermometer.TEMP_CHANGED, onTempChanged);
  }
  protected function onTempChanged(event:Event):void {
    var t:Thermometer = Thermometer(event.target);
    var ctx:ColorTransform = new ColorTransform();
    var hotness:Number = mapRange(t.temp, 60, 100);
    ctx.redMultiplier = hotness;
    ctx.greenMultiplier = 1 - hotness;
    this.transform.colorTransform = ctx;
  }
  //map a range of values from min->max to 0->1 (and clamp)
  protected function mapRange(value:Number, min:Number, max:Number):Number {
    return Math.min(1, Math.max(0, (value - min) / (max - min)));
  }
}
import flash.events.*;
import flash.utils.setInterval;
class Thermometer extends EventDispatcher {
  protected var _temp:Number = 72;
  public static const TEMP_CHANGED:String = "tempChanged";

  public function set temp(newTemp:Number):void {
    _temp = newTemp;
    dispatchEvent(new Event(TEMP_CHANGED));
```

```
    }
  public function get temp():Number {
    return _temp;
  }
  internal function debugSimulateCrazyWeather():void {
    //tweak the weather to trend continuously higher (to see our handiwork)
    setInterval(function():void {temp += 3 * (Math.random() - 0.4);}, 500);
  }
}
```

The event is sent to both the example class and the new `TempWarning` class. Notice that the code for `Thermometer` didn't have to change at all. This will become a crucial feature as your applications become more complex. You'll find that, unlike using callback functions or storing a reference to all the objects that need to be informed, using events to communicate between objects is a scalable solution.

Using EventDispatcher by Composition

`IEventDispatcher`, an interface found in the `flash.events` package, identifies a class as being capable of dispatching events. `EventDispatcher` implements `IEventDispatcher`, which lets you make a custom class an event dispatcher without actually subclassing `EventDispatcher`. Say you have a class called `Sprocket` that must extend a custom class written by your client called `Widget` (`Widget` can be any class that does not inherit from `EventDispatcher`). So you would have this:

```
class Sprocket extends Widget {
  //...
}
```

How would you add the ability to dispatch events to `Sprocket` (assuming that you cannot go back and edit `Widget`)? Remember that each class can extend only one other class.

The answer is that you add the event dispatching behavior by composition. That is, you compose the class to include an event dispatcher. By combining this with the `IEventDispatcher`, you can get the same functionality as an `EventDispatcher` by implementing the methods and passing them through onto the internal `EventDispatcher` object. You don't have to understand all of these methods to use them; simply pass them off to the internal `EventDispatcher` instance and let it worry about them.

```
package {
  import flash.events.Event;
  import flash.events.EventDispatcher;
  import flash.events.IEventDispatcher;

  public class Sprocket extends Widget implements IEventDispatcher {
    protected var dispatcher:EventDispatcher;
    public function Sprocket() {
      dispatcher = new EventDispatcher();
    }
```

```
        public function addEventListener(type:String,
                                         listener:Function,
                                         useCapture:Boolean=false,
                                         priority:int=0,
                                         useWeakReference:Boolean=false):void {
          return dispatcher.addEventListener(type, listener, useCapture,
                                             priority, useWeakReference);
        }
        public function removeEventListener(type:String,
                                            listener:Function,
                                            useCapture:Boolean=false):void {
          return dispatcher.removeEventListener(type, listener, useCapture);
        }
        public function dispatchEvent(event:Event):Boolean {
          return dispatcher.dispatchEvent(event);
        }
        public function hasEventListener(type:String):Boolean {
          return dispatcher.hasEventListener(type);
        }
        public function willTrigger(type:String):Boolean {
          return dispatcher.willTrigger(type);
        }
      }
    }
```

As you can see, by doing this you can get a `Sprocket` with all the benefits of `Widgets` and `EventDispatchers`! Most of the time, you'll be able to extend `EventDispatcher`, but this technique is useful for those tight spots when you can't.

Working with Event Objects

When an event occurs in ActionScript 3.0 and a `dispatchEvent()` call is triggered, a message is sent to all the recipients of the event. That message comes in the form of an `Event` object. `flash.events.Event` is a class that contains any pertinent data relating to an occurrence during runtime.

The `Event` class is a base class for all other types of events. It can be subclassed and customized to fit your needs. The Flash Player API comes with a handful of these subclasses of `Event`, found in `flash.events`. Event subclasses may contain extra information unique to the kind of event they describe. For example, a `MouseEvent` contains information about the position of the mouse and the state of its buttons; a `ProgressEvent` contains the number of bytes in a file that have been loaded. You can find a list of all these classes in the AS3LR.

Before dispatching any message, you need to create a new `Event` object. Using the `Event` class is often sufficient. In general though, it's good form to subclass `Event` and create a customized event class that contains more data specific to your needs. You'll focus on creating a basic event for now.

Here's the function signature for the `Event` constructor:

```
    public function Event(type:String, bubbles:Boolean = false,
      cancelable:Boolean = false)
```

First is the `type` parameter. You've seen this before in this chapter — it's a string that describes the event taking place, such as `"click"` or `"load"`.

Next, `bubbles` is a flag you can pass to your event to indicate whether it should participate in the bubbling phase of the event flow. You'll see what the event flow is later in this chapter.

Finally, `canceleable` determines whether a default behavior can be canceled. This parameter is mostly useful for predefined events within the Flash Player API. Again, I'll talk about default behaviors later.

So to review, the `Event` constructor has a required `type` parameter and two less frequently used optional parameters. Let's create a new `Event` object simply by omitting these:

```
var event:Event = new Event("dance");
```

You just created a `dance` event! Firing this event is just as easy.

```
dispatchEvent(event);
```

Many times you'll see these lines combined:

```
dispatchEvent(new Event("dance"));
```

One more thing: it's not good practice to use raw strings such as `"dance"` in your code for a number of reasons. First, strings and other literals are difficult to change. If you needed to make this `"tango"` instead, you would have to search for every listener and make changes there as well. This method is also more prone to typos because you can't use code hinting. When working with constants, the compiler is able to check for errors at compile time, while string errors might pass through unchecked.

To avoid these problems, it's common to use a static constant for every event type you want to include. By convention, these constants are defined in the `Event` subclasses made for the event types. (Custom event *classes* can still be used for multiple event *types*.) If you don't want to create a subclass of the `Event` class, you can define the constant in the class that's dispatching the event instead:

```
public static const DANCE:String = "dance"; // add this to your class
dispatchEvent(new Event(DANCE));
```

However, for projects of any reasonable size, I recommend using custom event subclasses.

Adding and Removing Event Listeners

You've already taken a look at some examples that use `addEventDispatcher()`. Now I'll go over how listeners are added and removed in more detail. First, the following is the full method signature for `addEventListener()`:

```
function addEventListener(type:String,
                         listener:Function,
                         useCapture:Boolean = false,
                         priority:int = 0,
                         useWeakReference:Boolean = false):void
```

Whoa. There's a lot going on here.

- `type` — You can think of `type` as the frequency over which the event is broadcast, the same way a radio station uses a particular frequency to broadcast music. Most of the time, the `type` you pass in is stored as a static string in the `Event` class that you are using.

419

- `listener` — The function that will be invoked when the event is fired.
- `useCapture` — Determines whether you will listen for the event on the capture phase (`true`) or on the target and bubble phases (`false`). This is used only for events that use the event flow, which I discuss later in the chapter.
- `priority` — Even though all event listeners appear to execute simultaneously, there is a distinct order in which the code is executed. Sometimes event handlers conflict with each other. Say, for example, you have two different listener functions called `onLoad` in two different objects. If the first one changes the loaded data before the second one reads it, you might have an unexpected conflict that can be hard to troubleshoot. This is where the `priority` parameter comes in. This value is an integer that can be set only while adding the listener. A higher number listener is executed before lower numbers. If two listeners' priorities are the same, they're executed in the order they were added to the event dispatcher. In most cases, the default value, zero, can be used. If you find yourself relying on event priorities, it's time to restructure your code, because it's difficult to follow code that depends on something as obscure as event priorities scattered through the code.

  There may be cases where you want to abort the handling of events. For example, you might want to use a single event listener to validate data before allowing the event to propagate to other listeners, canceling the event if the data is bad. You can do this using the `stopImmediatePropogation()` method of the `Event` class. Calling this method cancels all event handlers that haven't already been invoked. This isn't permanent; the effects last only until the next time the event is dispatched.

- `useWeakReference` — Allows objects to be garbage-collected even if they are still subscribed to events. In many cases, you will not need to use this parameter (whose default is `false`), but for the sake of memory management, it's not a bad idea. However, using this parameter without thinking can cause its own problems, because in some cases objects can be removed by garbage collection while you still need them. Here, too, you should prefer meticulous removal of added event listeners to using this parameter as a crutch.

  ActionScript 3.0 introduces the concept of weak and strong memory references. Typically, an object is garbage-collected if there are no references to the object. That is, when no objects are using a variable, it is thrown out. Weak references allow you to reference an object, but the object is still eligible for garbage collection unless another object holds a strong reference to the object. By setting the `useWeakReference` flag to `true`, you create a weak link between the event broadcaster and the event listener. That way, if an event listener is deleted while still listening to the event broadcaster, the weak reference allows it to be garbage-collected. This helps to prevent memory leaks when you forget to unsubscribe from events. However, `useWeakReference` can cause premature collection if you don't keep your own references to the dispatchers, stopping events from functioning properly.

Removing an event listener is as easy as adding one, and the syntax is similar:

```
function removeEventListener(type:String,
                            listener:Function,
                            useCapture:Boolean  = false):void
```

as in:

```
dispatcher.addEventListener("eventType", onEvent);
dispatcher.removeEventListener("eventType", onEvent);
```

Note

If you set the `useCapture` flag to `true` when you add the event listener, you need to set it to `true` when removing it as well. ∎

420

Use `removeEventListener()` when you no longer want your listener to respond to events. You should always remove all event listeners to an object when they are no longer needed and when you're ready to destroy the object. If active event listeners are observing an object, Flash Player doesn't actually release the object — unless you've subscribed using weak references. In events that should happen only once, you can deregister the event listener right in the event handler. Example 20-3 shows what that looks like.

EXAMPLE 20-3 `http://actionscriptbible.com/ch20/ex3`

Deregistering a Single-Use Event

```
package {
  import com.actionscriptbible.Example;
  import flash.display.Loader;
  import flash.display.LoaderInfo;
  import flash.events.Event;
  import flash.net.URLRequest;
  public class ch20ex3 extends Example {
    public function ch20ex3() {
      var l:Loader = new Loader();
      l.load(new URLRequest("http://actionscriptbible.com/files/roger.gif"));
      l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoadComplete);
      l.y = 30;
      addChild(l);
    }
    protected function onLoadComplete(event:Event):void {
      event.target.removeEventListener(Event.COMPLETE, onLoadComplete);
      trace("done loading, listener removed");
    }
  }
}
```

The Event Flow

When objects in the display list dispatch events, the events take on a more sophisticated behavior that you can take advantage of. Instead of being fired directly from the dispatcher object, the event appears to come from every `DisplayObject` walking down from the `Stage` to the actual target that called `dispatchEvent()`. Then it's dispatched by the dispatcher itself, and finally it walks back the other way, appearing to come from the dispatcher's parent, then it's parent's parent, all the way back to the `Stage` again. This is called the *event flow*, and as strange as it may sound, it's a W3C recommendation for dealing with events.

Note

To clarify the phrase "objects in the display list," the event flow applies to `DisplayObjects` and their subclasses that are currently in a display list. That is, they have been added to another display object in the display list, or put another way, they can be found by walking down the display list from the stage. `DisplayObjects` that have been created but not added to the stage are not subject to the event flow. ∎

Even if you never use the other capabilities, the most common use of the event flow is to allow events dispatched by child display objects to appear to come from the parent objects. For example, say you had a sprite containing several buttons. Typically, listening for click events on all buttons requires you to call `addEventListener()` for each button. But, because events in AS3 are dispatched from every parent display object in the chain leading up to the target, you can listen for button clicks just once on the parent sprite. This tells you that the user has clicked in the parent sprite or one of its descendants.
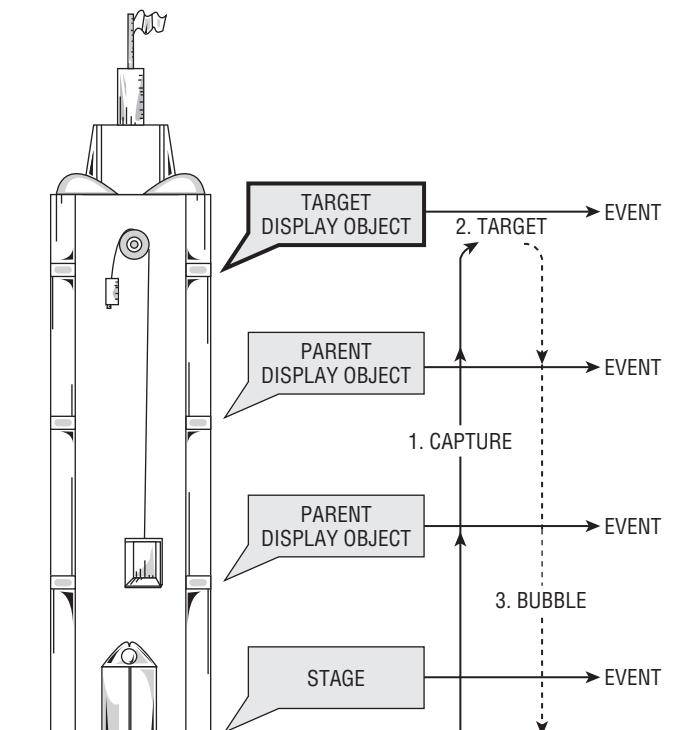
Let's look in detail at the three discrete phases that comprise the event flow.

The Phases of Event Flow

When the `dispatchEvent()` method is called from any `DisplayObject` in the display list, the event goes through three phases. These are called the *capture phase*, the *target phase*, and the *bubble phase*. You might think of an elevator where each floor is a parent display object on the way to the target. The elevator stops and dispatches an event on every floor it passes, as shown in Figure 20-3.

FIGURE 20-3

The three phases of event flow

Capture Phase

In the capture phase, Flash Player steps through every parent display object starting with the `Stage` and working up to the object that called `dispatchEvent()` and treats each one as though it were the event target. The effect is that the event is broadcast from every level of the display list that is a parent of the display object that calls `dispatchEvent()`. Display objects that are not ancestors of the target object do not dispatch events. To listen for events in the capture phase, you have to set the `useCapture` flag to `true` when adding the event listener.

Target Phase

During this phase, the event is dispatched directly from the target object. This is the phase you should be familiar with from the examples presented thus far. It is the only phase that events partake in for objects not in the display list.

Bubble Phase

Optionally, the event may bubble back up, dispatching from each parent display object until the stage is reached. Whether an event bubbles is set when creating a new `Event` object. The default setting depends on the class of the `Event`, but it is used for mouse and key events as well as other events dispatched by the Flash Player API. In general, I find the bubble phase is useful because you are guaranteed that the target phase event dispatcher has already had a chance to deal with the event by the time it is bubbling.

Note

By default, the target and bubble phases are listened for when calling `addEventListener()`. By setting `useCapture` to `true`, the capture phase is used instead. If you want to use all three phases, you have to add two separate event listeners: one with `useCapture` set to `true` and one with it set to `false`. ∎

Because the stage is the parent object for all display objects in the display list, events that use event flow always pass through the stage. Therefore, it's possible to listen for *any* event coming from within the display list by adding event listeners to the stage. This is especially helpful for keyboard events, which you often want to listen to globally. Remember that you can always use the `type` property of an event to filter your responses from within an event handler.

The current phase of an event in the flow can be determined from the `eventPhase` property of the `Event` object. The values returned are one of the following constants: `EventPhase.CAPTURING_PHASE`, `EventPhase.AT_TARGET`, or `EventPhase.BUBBLING_PHASE`. You can use a check like the one that follows to determine where in the event flow an event is and respond differently depending on which phase it's in. If you put this listener on the stage, you can see every event fired by any `DisplayObjects` in the display list.

```
private function onEvent(event:Event):void {
    if (event.eventPhase == EventPhase.CAPTURING_PHASE) {
        trace("Capture Phase");
        trace("Event is at " + event.currentTarget);
    }
    else if (event.eventPhase == EventPhase.AT_TARGET) {
        trace("Target Phase");
        trace("Event target is " + event.target);
    }
```

```
          else if (event.eventPhase == EventPhase.BUBBLING_PHASE) {
            trace("Bubble Phase");
            trace("Event is at " + event.currentTarget);
          }
        }
      }
```

Notice that you're using both `event.target` and a property I haven't covered yet: `event.currentTarget`. This property returns the display object that is currently being targeted within the event flow, whereas the `target` property remains the same and is always the object from which the event was dispatched.

At any time, the events in the event flow might be terminated by an event handler by using `stopPropagation()` or `stopImmediatePropagation()`. Overriding propagation can be useful to prevent other listeners from receiving the event, for example, if you have a modal dialog box open, you might handle any input in the dialog box and then stop the events, so that other display objects up the chain don't receive the input.

Event Flow in Action

The best way to understand the event flow is to see it in action. In Example 20-4, you create a set of several buttons contained within a button container. To emphasize how the event bubbling affects all the parent classes, place the button container inside a larger UI container, which you in turn place on the stage. Even though you'll create three buttons, only one listener is needed to handle all the click events.

EXAMPLE 20-4 `http://actionscriptbible.com/ch20/ex4`

The Event Flow

```
package {
  import com.actionscriptbible.Example;
  import flash.display.*;
  import flash.events.MouseEvent;
  import flash.text.TextField;
  public class ch20ex4 extends Example {
    public function ch20ex4() {
      var uiContainer:Sprite = new Sprite();
      uiContainer.x = 200;
      uiContainer.name = "uiContainer";
      addChild(uiContainer);

      //create the button container and add it to the UI container.
      var buttonContainer:Sprite = new Sprite();
      buttonContainer.graphics.beginFill(0x666666);
      buttonContainer.graphics.drawRect(0, 0, 250, 50);
      buttonContainer.name = "buttonContainer";
      buttonContainer.y = 20;
      uiContainer.addChild(buttonContainer);

      //create the UI label and add it to the UI container.
      var uiLabel:TextField = new TextField();
      uiLabel.name = "uiLabel";
      uiLabel.text = "Audio Controls";
```

```
      uiLabel.selectable = false;
      uiLabel.width = 80;
      uiLabel.height = 20;
      uiContainer.addChild(uiLabel);

      //create three buttons and add them to the button container.
      var stopButton:Button = new Button("Stop");
      stopButton.x = 10;
      stopButton.y = 10;
      buttonContainer.addChild(stopButton);

      var playButton:Button = new Button("Play");
      playButton.x = 90;
      playButton.y = 10;
      buttonContainer.addChild(playButton);

      var pauseButton:Button = new Button("Pause");
      pauseButton.x = 170;
      pauseButton.y = 10;
      buttonContainer.addChild(pauseButton);

      uiContainer.addEventListener(MouseEvent.CLICK, onClick);
    }
    private function onClick(event:MouseEvent):void {
      trace("\nClick received.");
      trace("Event Target:", DisplayObject(event.target).name);
      trace("Current Target:", DisplayObject(event.currentTarget).name);
    }
  }
}
import flash.display.Sprite;
import flash.text.TextField;
class Button extends Sprite {
  private var labelField:TextField;
  public function Button (label:String = "button") {
    //draw the background for the button.
    graphics.beginFill(0x3366CC);
    graphics.drawRect(0, 0, 70, 30);
    //store the label as the button's name.
    name = label;
    //create a TextField to display the button label.
    labelField = new TextField();
    //ensure clicks are sent from labelField rather than the button.
    labelField.mouseEnabled = false;
    labelField.selectable = false;
    labelField.text = label;
    labelField.x = 10;
    labelField.y = 10;
    labelField.width = 80;
    labelField.height = 20;
    addChild(labelField);
  }
}
```

Notice that the only place where you added an event listener was on `uiContainer`. The UI container is near the top of the display list ancestry because it contains the button container, which in turn contains all the buttons. Because every parent display object dispatches the events, this is the only listener where you need to capture clicks from the buttons. In fact, you may notice that this captures clicks from not only the buttons, but from the `uiLabel TextField` and from the gray `buttonContainer` as well.

Clicking each of the buttons triggers the `onClick` handler and produces the following results:

```
Click received.
Event Target: Stop
Current Target: uiContainer
Click received.
Event Target: Play
Current Target: uiContainer
Click received.
Event Target: Pause
Current Target: uiContainer
```

Again, you're using the event's `target` property to get the original display object that received the click and the `currentTarget` property to get the display object that's being listened to (in this case, `uiContainer`).

Some properties of `InteractiveObject` and `DisplayObjectContainer` (subclasses of `DisplayObject`) can help fine-tune events in the event flow. `mouseEnabled` controls whether the instance is permitted to fire mouse events, and `mouseChildren` controls whether the instance's children are permitted to. These are particularly important when creating custom cursors (your clicks must go through the cursor and interact with the scene below it) and buttons with labels. (If you are using `target` instead of `currentTarget`, clicks on a label inside a button fire events whose target is the label.) You can see these and the event flow interact at `http://dispatchevent.org/roger/hand-me-that-cursor-would-you/`.

Preventing Default Behaviors

Many events are so commonplace in ActionScript programming that they are taken for granted. For example, when a user inputs text into a text field, the letters appear on-screen. You may not realize it, but that is the result of an event handler being invoked behind the scenes. Handlers like this one — where there is an expected, predictable result — are called *default behaviors*.

Only events defined in the Flash Player API have default behaviors associated with them. You can't create a custom event that has a default behavior in Flash Player.

The event framework allows you to stop some of these default behaviors from occurring. Each `Event` object has a property called `cancelable`, a flag set in the constructor that lets you know whether the default behavior can be canceled. If this flag is set to `true`, as it is by default in `TextEvent.TEXT_INPUT`, the default behavior can be prevented using the method `preventDefault()`. The `Event`'s `isDefaultPrevented` property will let you know whether default behavior has been canceled. Preventing default actions is great for intercepting text entry, mouse scrolling, and other user inputs and repurposing them for your own devices.

Summary

- Events notify observing objects of interesting occurrences and give those observers a chance to react.

- Events are used throughout the Flash Player API, especially for user input, and asynchronous operations such as network access.

- Flash Player uses a lot of its own events, but you can also create and use your own.

- A *dispatcher* fires *event objects* of a specific *type*. The event object knows its type and remembers the dispatcher as its *target*. The event then proceeds through the *phases* of the *event flow*, triggering the *listeners* of every object that has registered for the event.

- Events are dispatched by an `EventDispatcher` subclass, or, rarely, a class that implements `IEventDispatcher`.

- `DisplayObject` extends `EventDispatcher` so all display objects can fire events.

- Use `addEventListener()` on a dispatcher to listen for events; call `removeEventListener()` to stop listening.

- Event objects extend `Event`. Subclasses of `Event` can carry additional information about the event.

- At minimum, `Event` objects are differentiated by `type`, which is normally stored as a static `String` constant of the event class.

- Events that bubble can only be dispatched from objects currently on the display list.

- The *capture*, *target*, and *bubble* phases proceed in order. In most cases, the target phase is used, and sometimes the bubble phase. Capture phase and event priority are used sparingly.

- Events can be aborted out of the event flow, and some built-in events can be stopped from performing their default behavior.

Interactivity with the Mouse and Keyboard

The one consistent interface between most people and their computers is, as you know, the keyboard and mouse. Because you can use Flash Player for so many different kinds of applications, it has a flexible model for capturing the user's mouse and keyboard input. You can do anything from simply capturing clicks on buttons to accepting complex mouse gestures. Flash Player's input handling is closely tied to its event model and the display list. All mouse and keyboard input are available to ActionScript 3.0 code as events of type `MouseEvent` or `KeyboardEvent`. In this chapter, you'll learn how mouse and keyboard events are handled in general, investigate the various kinds of events Flash Player exposes, and see a variety of real-world input handling code. I'll also cover the focus model in Flash Player and creating custom context menus.

Mouse and Keyboard Event Handling

Every `InteractiveObject` can dispatch events when interacted with using the keyboard and mouse. This, of course, applies to the subclasses of `InteractiveObject`, which includes all `TextFields`, `Sprites`, `Loaders`, and the `Stage`; but not `Bitmaps` or `Shapes`.

You'll learn how to listen and respond to these events, determine which events are available, and discover how to combine them best throughout the course of this chapter. But first I have to show just how these events are broadcast.

Note
Some users use alternative input methods for assistive purposes or on other platforms. Most of these devices, however, emulate keyboard and mouse input. So whether the end user is using her fingers, a stylus, a DualShock 3 controller, eye-tracking hardware, or a trackball, Flash Player gets her input as mouse events. You don't need to take the terms "keyboard" and "mouse" literally. ∎
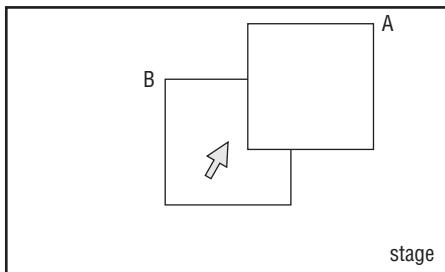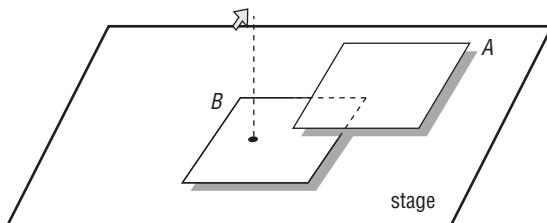
Finding the Target

Mouse and keyboard events are broadcast by display objects and bubble up the display list, as described in Chapter 20, "Events and the Event Flow." Events begin their dispatch at the target display object. For mouse events, this is typically the topmost display object under the mouse pointer that accepts mouse input. "Topmost" refers here to the stacking order or depth. When using display objects in three dimensions, z-depth is ignored; all that matters is the depth on the display list.

Think about positioning your screen flat on a table and dropping a pin down where your mouse cursor is. The object that the pin hits first is the target of your event. In Figure 21-1, I apply this test to determine that the target of mouse events will be display object B.

A simple layout, exploded out to show which object the pointer falls over

If you have a display object with a big hole in it, and the hole is truly empty (and not simply filled with a transparent or nearly transparent fill), your mouse and keyboard events can "fall through" this gap. This works great for vector shapes but not bitmaps, because "blank" areas of bitmaps are defined by transparent fills, which still catch the events.

Likewise, objects that don't react to the type of input you're giving them are "passed through" on Flash Player's quest from your mouse cursor to the target. Even if you create a big fat `Shape` that blocks out everything else and sits on top, because it is impervious to interactivity, clicks and other mouse events pass through it as if it didn't even exist.

Keyboard input, on the other hand, uses Flash Player's focus system to determine the target of the events. The object currently in focus dispatches keyboard events, regardless of mouse position. Although mouse and keyboard events determine their targets differently, once they're dispatched, both bubble.

Bubbling and Nested Clips
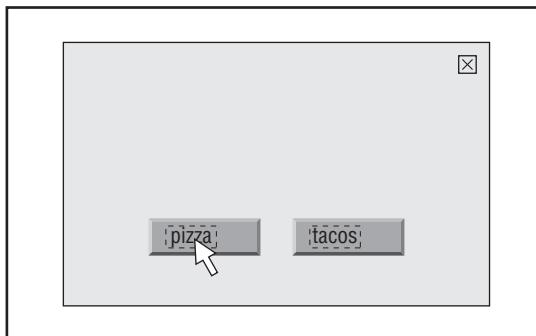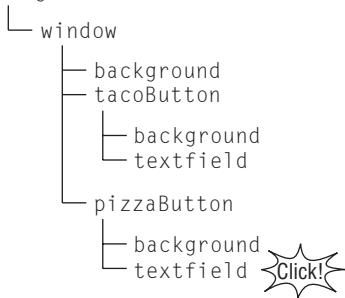
Almost all keyboard and mouse events bubble. (The exceptions are mouse rollover and rollout.) Once the event is triggered on the target object, it travels up the display list from the target object, to its container, and so on, until it hits the stage. This gives you multiple opportunities to react to user input. If you interpret the input event once and determine that parents of the `currentTarget` display object shouldn't hear about this event, you can call `stopPropagation()` on the event.

The nesting of display objects can affect how input events flow through the display list. In Figure 21-2, for example, the user has clicked on the Pizza button. Because the mouse cursor is over the `TextField` used to display the text `pizza`, that `TextField` is the target of the event. If you are subscribed to its parent, the `pizzaButton` display object, you'll receive the event as well, but only during the bubble phase. Likewise, the `window` object will receive the event, but not the `background` object, because it's a sibling of `pizzaButton`, even though it's directly underneath `pizzaButton`.

Input event bubbling

You can use four properties of `InteractiveObject` to effect tighter control on keyboard and mouse events, especially when display objects are nested:

- `mouseEnabled, keyboardEnabled` — Determine whether mouse or keyboard events can be triggered on the object.

- `mouseChildren, keyboardChildren` — Determine whether children of the object can receive mouse or keyboard events. If set to `false`, the object becomes "opaque" to mouse or keyboard events.

All these properties default to `true`. By changing them, you can impact how Flash Player determines the target of a mouse or keyboard input event. By setting `mouseEnabled` to `false` on a display object, you render it invisible to mouse events so that Flash Player "passes through" it when searching for the target of a mouse event. By setting `mouseChildren` to `false` on a display object, you "flatten" it — both the display object and its children are considered one potential target. Set both to `true` and the display object and its contents become completely non-reactive, ghosts to mouse input. This is particularly useful if you have display-list-based visual effects in the extreme foreground, or if you create a custom cursor, as in Example 21-8 later in the chapter.
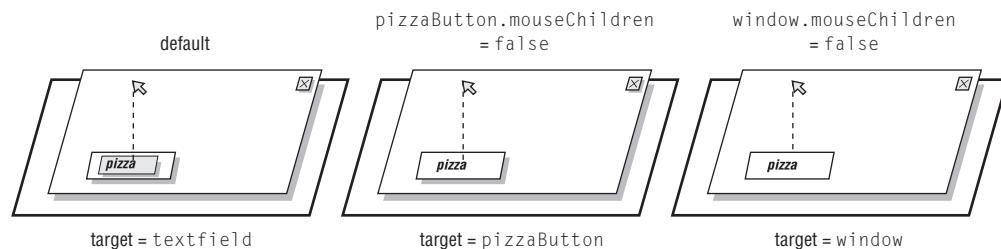
In Figure 21-3, I compare clicks inside the Pizza button's `TextField` when all objects' `mouseChildren` properties are set to `true`, when `pizzaButton`'s `mouseChildren` property is set to `false`, and when `window`'s `mouseChildren` property is set to `false`. In all cases, a click is made over the same place. The click is considered to have occurred on the innermost reachable object, which changes due to the "flattening" effect of `mouseChildren`.

Effects of `mouseChildren` with nested display objects

Of course, these effects apply to keyboard events as well as mouse events, and with the two sets of properties, you can control them differently.

Listening for All Events

Because input events that aren't canceled find themselves eventually at the stage, you can capture all events of a given kind by subscribing to the stage. Listening to the stage is especially useful for games, where the player's key and mouse input should be interpreted to affect her character regardless of the position of the cursor or focus.

Mouse Interactions

Mouse input is translated by Flash Player into events of type `MouseEvent`. Mouse input comes in many types, differentiated by the event's `type` property. A quick summary of these events is shown in Table 21-1.

Although these events may be self-explanatory, I'll take some time in the following sections to show exactly how the various mouse events work in context.

As you may have noticed, some of these events overlap each other. For example, a click (`MouseEvent.CLICK`) is the aggregate of the user depressing the mouse (`MOUSE_DOWN`) and releasing it (`MOUSE_UP`) in rapid succession, possibly with a small tolerance of mouse motion (`MOUSE_MOVE`) as well. If you listen for all these events, you will receive all of them. This allows you to make your mouse input handling quick and simple or as nuanced as you like.

The `MouseEvent` class defines a few properties that apply to mouse input. Some of these properties are related to specific types of mouse input and will remain undefined for other types of event that

use `MouseEvent`. Among the more important properties are four `Number`s that describe the cursor's location when the event took place:

- `localX`, `localY` — The cursor's x and y position relative to the target of the mouse activity
- `stageX`, `stageY` — The cursor's global x and y position

Note that when handling a mouse event called `event`, `event.localX` should be equal to `event.target.mouseX`, and `event.stageX` should be equal to `stage.mouseX` (and likewise with the y properties). Recall from Chapter 14, "Visual Programming with the Display List," that it's easy to translate between coordinate spaces; having the position defined for both spaces is a convenience.

TABLE 21-1

Mouse Events

Event Name	Description
`MouseEvent.CLICK`	The main button of the mouse was clicked.
`MouseEvent.DOUBLE_CLICK`	The main button of the mouse was double-clicked.
`MouseEvent.MOUSE_DOWN`	The main button of the mouse was depressed.
`MouseEvent.MOUSE_UP`	The main button of the mouse was released.
`MouseEvent.MOUSE_MOVE`	The position of the mouse cursor moved.
`MouseEvent.MOUSE_OVER`	The cursor moved over the display object.
`MouseEvent.MOUSE_OUT`	The cursor moved off the display object.
`MouseEvent.ROLL_OVER`	The cursor moved over the display object from outside it and its children.
`MouseEvent.ROLL_OUT`	The cursor moved off the display object and any of its children.
`MouseEvent.MOUSE_WHEEL`	The mouse wheel was rotated.

Tip

On rare occasions, I've experienced that the event object's location properties are incorrect. If your mouse position code seems to be error-prone in mouse event handlers, try using the `mouseX` and `mouseY` properties of the `Stage` instead; just ensure that the code always has access to the `stage` object, or you might run into an error. ■

Clicking

Flash Player makes it easy to detect clicks on objects. The user's operating system interprets the time between the mouse button being depressed and released to determine whether these two events constitute a click. You don't have to worry about just how the click occurred. Remember that this and other mouse events are affected by the `mouseEnabled` and `mouseChildren` properties, not just for the objects you click but their ancestors on the display list.

Detecting a click on any `InteractiveObject` is trivial, as Example 21-1 shows.

EXAMPLE 21-1 `http://actionscriptbible.com/ch21/ex1`

Detecting a Click

```
package {
  import com.actionscriptbible.Example;
  import flash.events.MouseEvent;
  public class ch21ex1 extends Example {
    public function ch21ex1() {
      var circ:ClickableCircle = new ClickableCircle();
      circ.name = "Circle";
      circ.x = circ.y = 100;
      addChild(circ);

      circ.addEventListener(MouseEvent.CLICK, onClick);
    }
    protected function onClick(event:MouseEvent):void {
      trace(event.target.name + " clicked at " +
            event.localX + "," + event.localY);
    }
  }
}
import flash.display.Sprite;
class ClickableCircle extends Sprite {
  public function ClickableCircle(color:uint = 0, size:Number = 50) {
    graphics.beginFill(color, 0.25);
    graphics.drawCircle(0, 0, size);
    graphics.endFill();
  }
}
```

Running this example and clicking the circle prints the name of the display object, along with the coordinates where you clicked.

Button Mode and the Hand Cursor

If you run Example 21-1, you will notice that although the `ClickableCircle` is indeed clickable, it doesn't indicate as much. It doesn't react to your mouse moving over it. If you want a hand cursor to appear in place of the normal pointer cursor, you can do so by indicating to Flash Player that the display object is a button. You can do this in two ways: make it a subclass of `SimpleButton`, or set its `buttonMode` property.

As far as Flash Player is concerned, being a button means a few important things. One, a button is included in the tabbing order, so you can press the Tab key to put focus on the button. Two, if you are focused on the button, pressing the spacebar simulates a click. These two effects ensure that the user interface is accessible to the keyboard and assistive devices. You can always customize

the focus and tabbing effects in more detail with the display object's `tabEnabled`, `tabOrder`, and `focusRect` properties. Three, buttons change your cursor to the system's hand cursor when hovered over. These effects are shown in Example 21-2.

EXAMPLE 21-2   http://actionscriptbible.com/ch21/ex2

Button Mode

```
package {
  import com.actionscriptbible.Example;
  import flash.events.MouseEvent;
  public class ch21ex2 extends Example {
    public function ch21ex2() {
      var circ:ClickableCircle = new ClickableCircle();
      circ.name = "Circle";
      circ.x = circ.y = 100;
      addChild(circ);
      circ.addEventListener(MouseEvent.CLICK, onClick);
    }
    protected function onClick(event:MouseEvent):void {
      trace(event.target.name + " clicked at " +
            event.localX + "," + event.localY);
    }
  }
}
import flash.display.Sprite;
class ClickableCircle extends Sprite {
  public function ClickableCircle(color:uint = 0, size:Number = 50) {
    graphics.beginFill(color, 0.25);
    graphics.drawCircle(0, 0, size);
    graphics.endFill();
    buttonMode = true;
  }
}
```

Here you've added one line to Example 21-1 to enable button mode. In `Sprite` and other `InteractiveObject` subclasses, `buttonMode` defaults to `false`, except of course for `SimpleButton`, where it defaults to `true`. If you run this code, you see all three effects mentioned in action. You might also notice that, as promised, where there is no fill in a display object, it does not react to clicks. If you put your mouse just outside the edge of the circle, even though you may be inside the display object's bounding box (which you can see by tabbing to it), you won't trigger the hand cursor or any click events.

Complex Clicking

The `MouseEvent.CLICK` event is triggered when the user's primary mouse button is clicked. Whether the device is an actual mouse or something like a tablet or a gamepad, how many buttons

the pointing device has and so on is of no consequence. Which button is considered the primary button is also up to the user and his operating system.

The secondary mouse button — usually the right button — does not trigger a mouse event in the Flash Player API. The secondary mouse button is reserved for displaying a standard context menu, which Flash Player retains some control over but you can indeed customize. Further mouse buttons, like a middle button, are not supported in Flash Player.

Note

In the AIR runtimes, you will have much more control over and access to mouse input. For example, the middle and right mouse buttons are supported. If you are developing for AIR, check the AS3LR for these additional events. ∎

Keyboard Modifiers

You can detect whether the user holds down certain keyboard modifier keys while clicking. The `MouseEvent` object defines several `Boolean` properties you can use to detect these keys:

- `shiftKey` — Whether the Shift key is held down
- `ctrlKey` — On PCs, whether the Control key is held down; on Macs, whether either of the Control or Command keys is held down
- `altKey` — On PCs, whether the Alt key is held down; on Macs, ignored

You'll use modifier keys in Example 21-6 to create and drag a copy of a display object instead of dragging the original object.

Double-Clicking

Flash Player even tells you if the user double-clicks a display object. Although this is off by default, simply set the display object's `doubleClickEnabled` to enable it, as shown in Example 21-3.

EXAMPLE 21-3 `http://actionscriptbible.com/ch21/ex3`

Detecting a Double-Click

```
package {
  import com.actionscriptbible.Example;
  import flash.events.MouseEvent;

  public class ch21ex3 extends Example {
    public function ch21ex3() {
      var circ:ClickableCircle = new ClickableCircle();
      circ.name = "Circle";
      circ.x = circ.y = 100;
      addChild(circ);
      circ.addEventListener(MouseEvent.CLICK, onClick);
    }
```

```
    protected function onClick(event:MouseEvent):void {
      trace(event.target.name + " clicked at " +
            event.localX + "," + event.localY);
    }
  }
}
import flash.display.Sprite;
import flash.events.MouseEvent;
class ClickableCircle extends Sprite {
  public function ClickableCircle(color:uint = 0, size:Number = 50) {
    graphics.beginFill(color, 0.25);
    graphics.drawCircle(0, 0, size);
    graphics.endFill();
    buttonMode = true;
    doubleClickEnabled = true;
    addEventListener(MouseEvent.DOUBLE_CLICK, onDoubleClick);
  }
  protected function onDoubleClick(event:MouseEvent):void {
    alpha *= 0.5;
    if (alpha < 0.1) alpha = 1;
  }
}
```

In this example, you've elected to have the ClickableCircle handle its own double-click event because you want this to be an effect of all ClickableCircles. If you try running the example, you'll see that the click event is still broadcast. In fact, when you double-click an object, the events are dispatched in order:

```
mouseDown
mouseUp
click
mouseDown
mouseUp
doubleClick
```

So you can still react to behavior on a lower level while using high-level events like MouseEvent .CLICK and MouseEvent.DOUBLE_CLICK.

Rollovers

Flash Player makes it easy to react to rollovers or hovering. You can use rollovers to change the appearance of a display object as the mouse hovers over it to emphasize the fact that it's active.

The SimpleButton class has this behavior built in. It switches between its upState, overState, and downState display objects automatically as the mouse leaves, enters, and depresses the button.

You can easily add this behavior to any InteractiveObject by listening to the MouseEvent .ROLL_OVER and MouseEvent.ROLL_OUT events, as Example 21-4 shows.

437

EXAMPLE 21-4 http://actionscriptbible.com/ch21/ex4

Reacting to Mouse Hover

```
package {
  import flash.display.Sprite;
  public class ch21ex4 extends Sprite {
    public function ch21ex4() {
      var circ:ClickableCircle = new ClickableCircle();
      circ.x = circ.y = 100;
      addChild(circ);
    }
  }
}
import flash.display.Sprite;
import flash.events.MouseEvent;
class ClickableCircle extends Sprite {
  public function ClickableCircle(color:uint = 0, size:Number = 50) {
    graphics.beginFill(color, 0.25);
    graphics.drawCircle(0, 0, size);
    graphics.endFill();
    addEventListener(MouseEvent.ROLL_OVER, onRollOver);
    addEventListener(MouseEvent.ROLL_OUT, onRollOut);
    onRollOut(null); //start in the "up"/not hovered state.
  }
  protected function onRollOver(event:MouseEvent):void {
    alpha = 1;
  }
  protected function onRollOut(event:MouseEvent):void {
    alpha = 0.5;
  }
}
```

Two sets of events deal with hovering: MOUSE_OVER, MOUSE_OUT, ROLL_OVER, and ROLL_OUT. The difference between these is subtle. The ROLL_OVER and ROLL_OUT events apply to a display object and its children, but the MOUSE_OVER and MOUSE_OUT events only apply to the display object. When moving your mouse from an inner display object to an outer display object, MOUSE_OUT is dispatched on the inner object as MOUSE_OVER is dispatched on the outer object. The difference might not be noticeable, however, because these events bubble, and the MOUSE_OUT event is followed up closely by a MOUSE_OVER event, although their targets are different.

In Example 21-5, you cancel bubbling to see just how the events differ. You'll notice that when you use MOUSE_OVER and MOUSE_OUT, as the blue circles on the right do, only the one specific display object under the mouse receives events, so the parent circle's stroke disappears when you mouse onto its child. It's best experimented with interactively, so run the example if you have a chance.

EXAMPLE 21-5 `http://actionscriptbible.com/ch21/ex5`

MOUSE_OVER versus ROLL_OVER

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  public class ch21ex5 extends Sprite {
    public function ch21ex5() {
      //use ROLL_OVER and ROLL_OUT on the left (red)
      var a:NestedCircles = new NestedCircles(true, 0xff0000);
      //use MOUSE_OVER and MOUSE_OUT on the right (blue)
      var b:NestedCircles = new NestedCircles(false, 0x0000ff);
      a.x = 100;
      b.x = 250;
      a.y = b.y = 150;
      addChild(a); addChild(b);
    }
  }
}
import flash.display.*;
import flash.events.MouseEvent;
class NestedCircles extends Sprite {
  public var child:NestedCircles;
  protected var stroke:Shape;
  public function NestedCircles(useRoll:Boolean, color:uint = 0,
                                size:Number = 60, isChild:Boolean = false) {
    graphics.beginFill(color, 0.25);
    graphics.drawCircle(0, 0, size);
    graphics.endFill();
    stroke = new Shape();
    addChild(stroke);
    stroke.graphics.lineStyle(5, 0xffff00);
    stroke.graphics.drawCircle(0, 0, size);
    stroke.visible = false;
    if (useRoll) {
      addEventListener(MouseEvent.ROLL_OVER, handler);
      addEventListener(MouseEvent.ROLL_OUT, handler);
    } else {
      addEventListener(MouseEvent.MOUSE_OVER, handler);
      addEventListener(MouseEvent.MOUSE_OUT, handler);
    }
    if (!isChild) {
      child = new NestedCircles(useRoll, color, size/2, true);
      addChild(child);
      child.y = -size;
    }
  }
  protected function handler(event:MouseEvent):void {
    switch (event.type) {
```

```
      case MouseEvent.MOUSE_OUT:
      case MouseEvent.ROLL_OUT:
        stroke.visible = false;
        event.stopPropagation();
        break;
      case MouseEvent.MOUSE_OVER:
      case MouseEvent.ROLL_OVER:
        stroke.visible = true;
        event.stopPropagation();
        break;
    }
  }
}
```

In most cases, you'll probably want to stick to `MouseEvent.ROLL_OVER` and `MouseEvent.ROLL_OUT`. And don't forget the time-saving `SimpleButton` option.

When using hover-related events, you're always moving the pointer from one object to another (even if one of these objects is the stage). Whereas the `target` property of the event object tells you the subject of this action — the `target` of a `MouseEvent.ROLL_OUT` event tells you where you've rolled off of — the `relatedObject` property tells you what the other party is — where you rolled off *onto*. Another example: you `MOUSE_OVER` a `target`, but your mouse was *previously* over the `relatedObject`.

Dragging

A `Sprite`'s `startDrag()` and `stopDrag()` methods are perfect for implementing drag-and-drop behavior, but you'll still need some event dispatching glue to put the whole thing together. Simply start dragging when the mouse button is pressed, and stop dragging when the mouse button is released. Example 21-6 shows how this works.

Drag-and-Drop

```
package {
  import flash.display.Sprite;
  public class ch21ex6 extends Sprite {
    public function ch21ex6() {
      var circ:DraggableCircle = new DraggableCircle();
      circ.x = circ.y = 100;
      addChild(circ);
    }
  }
}
```

```
import flash.display.Sprite;
import flash.events.MouseEvent;
class DraggableCircle extends Sprite {
  public function DraggableCircle() {
    graphics.beginFill(0, 0.5);
    graphics.drawCircle(0, 0, 50);
    graphics.endFill();
    addEventListener(MouseEvent.MOUSE_DOWN, onStartDrag);
    buttonMode = true;
  }
  protected function onStartDrag(event:MouseEvent):void {
    if (event && event.shiftKey) {
      cloneAndDrag();
    } else {
      startDrag();
      stage.addEventListener(MouseEvent.MOUSE_UP, onStopDrag);
    }
  }
  protected function onStopDrag(event:MouseEvent):void {
    stage.removeEventListener(MouseEvent.MOUSE_UP, onStopDrag);
    stopDrag();
  }
  protected function cloneAndDrag():void {
    var copy:DraggableCircle = new DraggableCircle();
    copy.x = this.x;
    copy.y = this.y;
    this.parent.addChild(copy);
    copy.onStartDrag(null);
  }
}
```

I've done one or two tricky things here, though. I've listened for `MouseEvent.MOUSE_UP` on the stage, so that I'll receive that event no matter what display object the mouse was over when the event occurred (provided nobody cancels the event prior to it bubbling up to the stage). I did this just in case you managed to have your mouse cursor escape the boundaries of the object in question. Because it's dragging, it *should* follow under your mouse cursor no matter what, but in its dragging travels it may pass under other display objects, or it might change shape or size slightly due to rollover effects, although it doesn't here. I just want to make sure that, no matter what, the next time you release the mouse button, the currently dragging object stops dragging. To complete this cycle, of course, I have to remove the event listener from the stage immediately when the mouse is released.

Additionally, I've looked at the state of the Shift key when the mouse is depressed. If you have the Shift key down, instead of dragging the circle, it makes a clone of the circle and drags that. Event handler methods are still normal methods, and they may be called directly as I have done here. However, if you pass an event object, it had better be valid. Because I sometimes pass `null` to the event handler, in its body I have to make sure I don't assume `event` is non-`null`.

Position Tracking and Cursors

You already have the tools in your arsenal to follow the mouse around. Simply listen for MouseEvent.MOUSE_MOVE on the stage, and you'll be updated with the mouse's movement. You can use this to display custom cursors and mouse trails, control a game character, or even just to stare at you creepily, as a big pair of eyes do in Example 21-7.

EXAMPLE 21-7 `http://actionscriptbible.com/ch21/ex7`

Following Mouse Movement

```
package {
  import flash.display.Sprite;
  public class ch21ex7 extends Sprite {
    public function ch21ex7() {
      var leftEye:Eye = new Eye(60);
      var rightEye:Eye = new Eye(60);
      leftEye.y = rightEye.y = stage.stageHeight/2;
      leftEye.x = stage.stageWidth * 1/3
      rightEye.x = stage.stageWidth * 2/3;
      addChild(leftEye);
      addChild(rightEye);
    }
  }
}
import flash.display.*;
import flash.events.*;
class Eye extends Sprite {
  protected var pupil:Shape;
  public function Eye(size:Number) {
    graphics.lineStyle(3);
    graphics.beginFill(0xffffff);
    graphics.drawCircle(0, 0, size);
    graphics.endFill();

    pupil = new Shape();
    addChild(pupil);
    pupil.graphics.lineStyle();
    pupil.graphics.beginFill(0x603030);
    pupil.graphics.drawCircle(3/4 * size, 0, size/4);

    scaleY = 2;
    addEventListener(Event.ADDED_TO_STAGE, onAddedToStage);
  }
  protected function onAddedToStage(event:Event):void {
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
  }
  protected function onMouseMove(event:MouseEvent):void {
    pupil.rotation = Math.atan2(this.mouseY, this.mouseX) / Math.PI * 180;
  }
}
```

Note that I had to wait for the `Event.ADDED_TO_STAGE` event before accessing `stage`. This property is not defined while the constructor executes, because the object hasn't been put in a display list yet. The application class doesn't have this restriction.

An interesting property of `MouseEvent.MOUSE_MOVE` events is that they fire off whenever the mouse moves, independently of the frame rate of the SWF. You can force Flash Player to draw a new frame as soon as ActionScript is done executing (without waiting for it to be time to draw the next frame as per the current frame rate). Simply call the `updateAfterEvent()` method of `Event` when it's imperative that there is no lag between your application's user interface and the mouse's location.

This is especially useful for creating your own cursors, as Example 21-8 shows. Although the Flash Player API uses the system's default pointer and hand cursors, any other kinds of cursors, or any custom cursors, must be implemented in ActionScript code. You can emulate a cursor by creating it in a display object at the highest depth and following the mouse's position with `MouseEvent.MOUSE_MOVE`.

EXAMPLE 21-8 `http://actionscriptbible.com/ch21/ex8`

Custom Cursors

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.net.URLRequest;
  import flash.ui.Mouse;
  public class ch21ex8 extends Sprite {
    protected var customCursor:Loader;
    public function ch21ex8() {
      customCursor = new Loader();
      var url:String = "http://actionscriptbible.com/files/spinner.swf";
      customCursor.load(new URLRequest(url));
      customCursor.mouseEnabled = false;
      customCursor.mouseChildren = false;
      addChild(customCursor);
      stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
      stage.addEventListener(Event.MOUSE_LEAVE, onMouseLeave);
    }
    protected function onMouseMove(event:MouseEvent):void {
      Mouse.hide();
      customCursor.visible = true;
      customCursor.x = stage.mouseX;
      customCursor.y = stage.mouseY;
      event.updateAfterEvent();
    }
    protected function onMouseLeave(event:Event):void {
      customCursor.visible = false;
    }
  }
}
```

In this example, I've gone a few steps further to ensure that the cursor acts as much like an OS cursor as possible. I have disabled mouse events on the cursor. Because the cursor is on the top and sticks to the mouse's position, it would otherwise become the target of all mouse events, stealing events that need to go to their intended targets. Turning off mouse events on the cursor allow clicks and mouse events to "fall through" it.

I've used the static method `Mouse.hide()`, which hides the system mouse cursor. If I didn't hide the original cursor, two cursors would appear, destroying the illusion of a custom cursor. When the mouse leaves Flash Player, the system restores the normal cursor. When you bring the mouse back over to Flash Player again, however, the cursor doesn't remember that it should be hidden. That's why I call `Mouse.hide()` every time the mouse moves, even though it may be overkill.

There is also a `Mouse.show()` method that makes the cursor visible again. You can use this if you only want to show a custom mouse cursor while hovered over a certain object, for example.

I've also used the `Stage` class's `Event.MOUSE_LEAVE` event. This event is broadcast when the mouse exits Flash Player. By hiding the custom cursor when the mouse leaves, you prevent the cursor from displaying near the edge when your real mouse cursor reappears, which breaks the illusion that the custom cursor *is* your cursor.

In Flash Player 10 and up, you can use additional kinds of standard OS cursors through the `Mouse` class, as listed in Table 21-2. Furthermore, you can invoke these cursors at will rather than relying on, for example, input `TextFields` and `SimpleButtons` that trigger cursors by default. To set the cursor type, simply assign the proper value to the static property `Mouse.cursor`. The available cursors are enumerated by static constants in the `flash.ui.MouseCursor` class.

TABLE 21-2

System Standard Cursors Supported in Flash Player 10

Cursor name	Cursor type
MouseCursor.AUTO	Whichever cursor is appropriate for the content under the mouse (default)
MouseCursor.ARROW	The default pointer arrow
MouseCursor.BUTTON	A button-clicking hand
MouseCursor.HAND	A dragging hand
MouseCursor.IBEAM	A vertical text-editing I-beam

Version

FP10. The `Mouse.cursor` property can only be used to set cursor types in Flash Player 10 and up. ∎

After you've set a special system cursor, you can restore the cursor to its default behavior by setting the cursor type to `AUTO`:

```
Mouse.cursor = MouseCursor.HAND; //set a custom cursor
Mouse.cursor = MouseCursor.AUTO; //reset to the default, contextual cursor
```

Blocking All Mouse Input

An interesting test of your mouse interaction skills is to block all clicks. You might need to do this if you present a modal dialog, which must be satisfied before any other user interaction can take place. Or, you might choose to prevent all input while transitioning between two states of your application — a quick and dirty way to prevent overlapping navigations that might throw your application out of sync.

Again, I've already covered the principles that you'll need to block all user input. If you want, take a moment and imagine how you would block all clicks. I suggest the following method: create a display object, fill it entirely with a transparent fill, ensure mouse events are enabled on it, and put it at the top of the display list. Ensure that it stays on top and stays the size of the stage. Optionally, capture all mouse events and stop their propagation to ensure even the stage doesn't receive mouse events. Because Flash Player looks for the topmost display object under the mouse to receive mouse events, this screener object catches them all. Example 21-9 shows how this works.

EXAMPLE 21-9 `http://actionscriptbible.com/ch21/ex9`

Blocking Clicks

```
package {
  import com.actionscriptbible.Example;
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  public class ch21ex9 extends Example {
    protected var screen:Sprite;
    public function ch21ex9() {
      makeTestButton();
    }
    protected function makeTestButton():void {
      var button:TestButton;
      button = new TestButton(100, 30, "Pizza");
      addChild(button); button.x = 10; button.y = 10;
      button.addEventListener(MouseEvent.CLICK, onPizzaClick);

      button = new TestButton(100, 30, "Tacos");
      addChild(button); button.x = 130; button.y = 10;
      button.addEventListener(MouseEvent.CLICK, onTacosClick);

      button = new TestButton(110, 30, "SCREEN ON!");
      addChild(button); button.x = 250; button.y = 10;
      button.addEventListener(MouseEvent.CLICK, onScreenClick);

      trace("\n\n\n");
    }
    protected function onPizzaClick(event:MouseEvent):void {
      trace("Pizza button clicked!");
    }
    protected function onTacosClick(event:MouseEvent):void {
      trace("Tacos button clicked!");
    }
```

continued

EXAMPLE 21-9   *(continued)*

```
    protected function onScreenClick(event:MouseEvent):void {
      trace("---- blocking events ----");
      screen = new Sprite();
      screen.graphics.beginFill(0, 0); //optionally, dim screen
      screen.graphics.drawRect(0, 0, stage.stageWidth, stage.stageHeight);
      //TODO: in a real application, add stage resize listeners
      screen.graphics.endFill();
      screen.mouseEnabled = true;
      addChild(screen); //TODO: ensure that screen stays on top
      //TODO: optionally, handle all events and cancel them
    }
  }
}
import flash.display.*;
import flash.text.*;
import flash.events.MouseEvent;
import flash.filters.BevelFilter;

class TestButton extends Sprite {
  protected var bg:Shape;
  protected var label:TextField;
  public function TestButton(w:Number, h:Number, labelText:String) {
    bg = new Shape();
    addChild(bg);
    bg.graphics.beginFill(0xa0a0a0);
    bg.graphics.drawRect(0, 0, w, h);
    label = new TextField();
    addChild(label);
    label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
      false, null, null, "center");
    label.width = w;
    label.height = h;
    label.text = labelText;
    label.y = (h - label.textHeight)/2 - 2;
    buttonMode = true;
    mouseChildren = false;
    addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    onMouseUp(null);
  }
  protected function onMouseDown(event:MouseEvent):void {
    bg.filters = [new BevelFilter(4, 225, 0xffffff, 0.5, 0, 0.5, 1, 1, 1, 3)];
    label.x++; label.y++;
  }
  protected function onMouseUp(event:MouseEvent):void {
    bg.filters = [new BevelFilter(-4, 225, 0xffffff, 0.5, 0, 0.5, 1, 1, 1, 3)];
    label.x--; label.y--;
  }
}
```

Although I'm getting a bit ahead of the game with filters and vector drawing, you can basically ignore the `TestButton` class. It's just there to demonstrate that mouse events are getting through. Once you click the third button, an invisible screen is constructed that blocks all mouse events (except from the stage, because we neglected to capture and cancel the events).

Mouse Wheel

You can capture input from the user's mouse wheel, if she has one, by listening for `MouseEvent.MOUSE_WHEEL` events. In event objects associated with this event, the `delta` property is populated with an `int` representing the number of lines and direction the user scrolled by. You may want to listen for this event on the stage because, like other mouse events, it is only broadcast by the topmost object found under the mouse cursor, and users may expect scroll wheel input to work even when the cursor is not directly over the target.

Caution

Scroll wheel support in the browser on Mac is not available before Flash Player 10.1, or in browsers that don't support Cocoa events like Safari 4. In non-compliant environments, you can use JavaScript to capture mouse wheel events and forward them to Flash with `ExternalInterface`. A free library that does this is SWFMacMouseWheel by Gabe Bucknall, found at `http://blog.pixelbreaker.com/flash/ swfmacmousewheel.` ∎

Keyboard Interactions

Keyboard input is easy to handle in Flash Player. There are only two events for keyboard input:

- `KeyboardEvent.KEY_DOWN` — A key is depressed
- `KeyboardEvent.KEY_UP` — A key is released

Keyboard input is dispatched by the focused `InteractiveObject` and bubbles up to the stage. I'll cover focus in depth later in the section titled "Focus."

When capturing key presses, you can choose to listen for either of these events. There's no real way to "cancel" a key press — once you've depressed the keyboard key, you've got little choice but to eventually let it up — so either event is sufficient for simply capturing keystrokes. Because each key's downstroke and upstroke are captured, you can react to simultaneous keystrokes without other events. Two keys pressed simultaneously results in four events: two `KEY_DOWN` and then two `KEY_UP`.

The way key events are structured, however, you have to write some code yourself if you want to answer the question "What keys are being held down right now?" in the general case, outside an event handler. The events don't answer this question; but by recording the individual downstrokes and upstrokes, you can remember if the key in question had previously been depressed and has not yet been released.

Interpreting Keypresses

Each `KeyboardEvent` object encodes the key to which it refers in two ways and makes both available to you through two properties: `keyCode` and `charCode`.

`keyCode` is a `uint` in which every key on the keyboard has a unique value. For instance, the 1 key on the numeric pad and the 1 key on the top row are different physical keys and have different

keyCodes. Conversely, both the letter a and the letter A are typed with the A key, which is the same physical key and has the same keyCode.

charCode is a uint that represents the character you typed in the current character set. The default character set is UTF-8, which is identical to ASCII for values 0–28 (including all English alphanumeric characters). Because charCode is concerned with the character, typing a and A produce two different characters and two different charCodes, even though they are typed with the same physical key. Typing 1 on any part of the keyboard results in a 1 character, so the charCode is the same.

You can convert one or more charCodes back into a String by passing them to the static method String.fromCharCode(). The method converts its arguments from UTF-8 character codes into the glyphs that they represent. To demonstrate, all three of these lines generate the same output, a Japanese ideograph found tattooed above butts across America:

```
trace("\u7f8e");
trace(String.fromCharCode(0x7f8e));
trace("美");
```

However, you're not likely to see such exotic character codes, especially if your keyboard layout is QWERTY or any Eurocentric layout. Keyboard events are for single keystrokes, and they take precedence over Input Method Editor (IME) input.

Example 21-10 is a simple test that both shows how to listen for keyboard events and demonstrates the differences between the two kinds of key properties.

EXAMPLE 21-10 http://actionscriptbible.com/ch21/ex10

Listening for Keyboard Events

```
package {
  import com.actionscriptbible.Example;
  import flash.events.KeyboardEvent;
  public class ch21ex10 extends Example {
    public function ch21ex10() {
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKey);
      stage.addEventListener(KeyboardEvent.KEY_UP, onKey);
      trace("event\t\tkeyCode\tcharCode");
      trace("------------------------------");
    }
    protected function onKey(event:KeyboardEvent):void {
      var char:String =
        (event.charCode > 31)? String.fromCharCode(event.charCode) : "n/a";
      trace(event.type + "\t\t" +
            event.keyCode + "\t\t" +
            event.charCode + "\t(" + char + ")");
    }
  }
}
```

Keys like Shift, Ctrl, and Esc have `keyCodes` but do not have associated `charCodes`. Thankfully, rather than remember what `keyCode` refers to which key, a handful of static properties are defined by the `Keyboard` class for common nonprinting keys. Some of these are the F or function keys like `Keyboard.F11`; cursor movement keys like `Keyboard.PAGE_UP`, `Keyboard.HOME`, and `Keyboard.END`; whitespace like `Keyboard.SPACE` and `Keyboard.TAB`; and toggle keys like `Keyboard.CAPS_LOCK` and `Keyboard.INSERT`. For the full list, consult the AS3LR.

In Example 21-11, I'll use some of the `Keyboard` constants with the `keyCode` property to move a character around a screen.

EXAMPLE 21-11   `http://actionscriptbible.com/ch21/ex11`

Using Keyboard Constants

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.KeyboardEvent;
  import flash.ui.Keyboard;
  [SWF(backgroundColor="0x000000", frameRate="4")]
  public class ch21ex11 extends Sprite {
    protected var hero:Hero;
    protected var keys:Array;
    protected const MAX_KEY:int = 128;
    public function ch21ex11() {
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKey);
      stage.addEventListener(KeyboardEvent.KEY_UP, onKey);
      keys = new Array(MAX_KEY);

      hero = new Hero();
      addChild(hero);
      hero.x = stage.stageWidth/2;
      hero.y = stage.stageHeight/2;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onKey(event:KeyboardEvent):void {
      if (event.keyCode >= MAX_KEY) return;
      keys[event.keyCode] = (event.type == KeyboardEvent.KEY_DOWN);
    }
    protected function onEnterFrame(event:Event):void {
      if (keys[Keyboard.UP]) hero.y -= hero.height;
      if (keys[Keyboard.DOWN]) hero.y += hero.height;
      if (keys[Keyboard.LEFT]) hero.x -= hero.width;
      if (keys[Keyboard.RIGHT]) hero.x += hero.width;
    }
  }
}
```

continued

EXAMPLE 21-11 *(continued)*

```
import flash.display.Shape;
class Hero extends Shape {
  public function Hero() {
    graphics.beginFill(0x10c010);
    graphics.drawRect(0, 0, 12, 30);
    graphics.endFill();
  }
}
```

I've also implemented a simple answer to "Which keys are being held down right now?" as described at the beginning of this section. Using the unique keyCodes as indices of an array, I set the values of the array to true when the key is depressed and false when the key is released, so that at any point in time key[anyCode] returns whether the key with code anyCode is depressed.

From there, I simply check the four directional keys every time it's time to draw a new frame, moving the character in the proper direction. Because keys can be held down simultaneously, you can hold down the right and down arrow keys, and the character will move diagonally as long as you have both held.

The end effect recalls ancient terminal graphics games like *Nethack*. Try it out!

Modifier Keys

Like mouse events, keyboard event objects tell you the state of the modifier keys on the keyboard at the time of the event. They use the same properties — altKey, ctrlKey, and shiftKey — for the same purposes. (You may review these in the earlier section titled "Complex clicking.") Although you can gain the same information by remembering the state of the modifier keys by watching for their KEY_DOWN events, accessing these modifier key properties of the event object is easier.

Related Events

You can use keyboard events to develop more interactive extensions of input text fields, such as those that automatically reformat your input as you type it or those that autocomplete your input. Be aware that changes may be made to a TextField's contents without key presses, such as by pasting with a context menu or by a voice recognition engine. Choose your events carefully. These two events dispatched by TextFields may prove useful to you:

- Event.CHANGE — Dispatched anytime the contents of the TextField change
- TextEvent.TEXT_INPUT — Dispatched when text is added to the TextField (but not when it is deleted)

IMEs

An Input Method Editor, or IME, allows you to enter characters not found on your keyboard by typing a series or combination of keys. For instance, using the default U.S. input method on Mac OS

X, you can hold down Opt and press e, and then release both keys and press e again (from here on I'll write this Opt-e, e) to generate the é character. Romaji-based Japanese IMEs on both OS X and Windows let you type Japanese text in Roman characters and then convert it; you can press h, i, t, o, spacebar to produce the character 人 for "person." In both examples, you've hit a complex sequence of keys on the keyboard, but the IME has converted those keystrokes after the fact into a single character. Many IMEs require significant direction from the user to choose the correct translation for the typed text.

`KeyboardEvent` events are only aware of the keystrokes you input before the IME does its work. However, you can interact with the IME using the `flash.system.IME` class.

Depending on the system and the IME software, you will have varying levels of access to the installed IMEs. Provided your system and IME are supported, you can request access to preconversion strings, set the conversion mode, send text to be converted, and accept the first available possible conversion.

The following code tests whether Flash Player supports IME interaction on the platform and whether the IME is active in the current context. The code also tells you what kind of text the IME produces after conversion.

```
if(Capabilities.hasIME && IME.enabled) {
  trace(IME.conversionMode);
}
```

Note that the user's IME functions perfectly well, converting keystrokes and inserting converted text, without intervention from Flash Player. The only purpose for using the `IME` class is to control the operation of the user's IME through code. In almost every case, you should leave the operation of the IME up to the user — although you may want to detect if one is active when listening to keyboard events so that you wait for the proper, converted text to arrive rather than listening keystroke by keystroke.

Focus

In graphical user interfaces, one object has focus at a given time. This focused object is considered the "active" object, and as such displays special behavior. In Flash Player, the focused object is the recipient of any keyboard events and can display some kind of focus indicator if it was focused by the keyboard. (By default, this indicator is a yellow rectangle surrounding the object.) Additionally, `SimpleButton`s display their "over" state when focused.

You can change Flash Player's focus with either the mouse or the Tab key. Focus can only be given to `InteractiveObject`s. You set focus with the mouse by clicking on an `InteractiveObject`, which becomes the new focus. Set can also focus with the keyboard by pressing Tab. The focus proceeds through `InteractiveObject`s as you repeatedly press Tab, repeating the cycle once it reaches the last object. The order these objects receive focus as you press Tab, or the *tabbing order*, is determined by Flash Player but can be overridden in ActionScript. As you change focus with either method, focus-related events are dispatched.

Finding or Setting the Current Focus

The focus of Flash Player is determined by the `Stage`'s `focus` property. This property is read-write, so you can set the focus by assigning a new `InteractiveObject` to it. When the `focus` property is `null`, no object has focus. Keyboard events are still received by the stage, however.

Setting and retrieving the focus can be useful to enable and disable UI controls. It's especially useful for input text fields, because the text field with focus is the one to receive keyboard input. In Example 21-12, I'll create a sample form and manipulate the focus to make it a bit more usable.

EXAMPLE 21-12 http://actionscriptbible.com/ch21/ex12

Manipulating Focus

```
package {
  import flash.display.Sprite;
  import flash.events.KeyboardEvent;
  import flash.events.MouseEvent;
  import flash.text.*;
  import flash.ui.Keyboard;
  public class ch21ex12 extends Sprite {
    protected var input:TextField;
    protected var submitButton:TestButton;
    protected var clearButton:TestButton;
    public function ch21ex12() {
      input = new TextField();
      input.type = TextFieldType.INPUT;
      input.defaultTextFormat = new TextFormat("_sans", 12, 0);
      input.width = 215; input.height = 20;
      input.border = true;
      addChild(input);
      input.x = 10; input.y = 10;
      input.addEventListener(KeyboardEvent.KEY_UP, onInputKey);

      submitButton = new TestButton(100, 30, "Submit");
      addChild(submitButton);
      submitButton.x = 10; submitButton.y = 40;
      submitButton.addEventListener(MouseEvent.CLICK, onSubmit);

      clearButton = new TestButton(100, 30, "Clear");
      addChild(clearButton);
      clearButton.x = 125; clearButton.y = 40;
      clearButton.addEventListener(MouseEvent.CLICK, onClear);

      stage.focus = input;
    }
    protected function onInputKey(event:KeyboardEvent):void {
      if (event.keyCode == Keyboard.ENTER) {
        onSubmit(null);
      }
    }
    protected function onSubmit(event:MouseEvent):void {
      stage.focus = null;
      submitButton.label.text = "Submitted!";
      input.mouseEnabled = input.tabEnabled = false;
      input.background = true;
      input.backgroundColor = 0xb0b0b0;
      clearButton.removeEventListener(MouseEvent.CLICK, onClear);
    }
```

```
      protected function onClear(event:MouseEvent):void {
        input.text = "";
        stage.focus = input;
      }
    }
  }
  import flash.display.*;
  import flash.text.*;
  class TestButton extends Sprite {
    public var label:TextField;
    public function TestButton(w:Number, h:Number, labelText:String) {
      graphics.lineStyle(0.5, 0, 0, true);
      graphics.beginFill(0xa0a0a0);
      graphics.drawRoundRect(0, 0, w, h, 8);
      label = new TextField();
      addChild(label);
      label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
        false, null, null, "center");
      label.width = w;
      label.height = h;
      label.text = labelText;
      label.y = (h - label.textHeight)/2 - 2;
      buttonMode = true;
      mouseChildren = false;
    }
  }
```

There are a few things I've done here with focus. I've started the application with focus already in the input text field. This means that as soon as you launch the application, you can start typing right away. (However, when the application is embedded in a web page, you'll still have to click somewhere in Flash Player to give it focus. Oh well.)

Pressing Enter in the input text field does the same thing as clicking the Submit button. Because the keyboard events are subscribed to `input`, only pressing Enter while it is focused triggers the submit process.

Clicking the Clear button not only clears `input`, but it focuses it so that you can immediately type in new input without clicking on it. (You removed focus from the text field when you clicked on the Clear button.)

Submitting the form disables the input text field. I've done so by removing focus from it and preventing you from returning focus to it. Because the user can only focus an object by clicking or tabbing into it, disabling mouse events and turning off `tabEnabled` cut off these two possibilities. More on `tabEnabled` later.

Focus Events

You got a preview of focus events in action with input text fields in Chapter 17, "Text, Styles, and Fonts," in the section "Interaction with TextField Events." This section also provides a good example

of how focus management can improve a user interface, which you might want to review now (`http://actionscriptbible.com/ch17/ex8`).

Not only `TextFields`, but all `InteractiveObjects` dispatch focus events when they receive and give up focus. Furthermore, they dispatch special events when they are about to relinquish focus via the mouse and keyboard. The focus events are

- `FocusEvent.FOCUS_IN` — The target received focus.
- `FocusEvent.FOCUS_OUT` — The target lost focus.
- `FocusEvent.KEY_FOCUS_CHANGE` — The target is about to lose focus from a keyboard command, such as the user pressing Tab.
- `FocusEvent.MOUSE_FOCUS_CHANGE` — The target is about to lose focus from a mouse event, such as the user clicking on another `InteractiveObject.`

The `KEY_FOCUS_CHANGE` and `MOUSE_FOCUS_CHANGE` events are cancelable, so you can handle and then cancel these events with `stopPropagation()` to prevent the user from taking focus away via one or both methods. You can trap the user into a text field until she answers correctly, for instance, although that would most likely constitute bad usability design.

Focus event objects, like mouse event objects, have a `relatedObject` property that refers to the other `InteractiveObject` related in an exchange of focus. When `target` refers to the object gaining focus, `relatedObject` refers to the object losing it, and vice versa.

Tabbing

As mentioned, Flash Player cycles through `InteractiveObjects` that receive keyboard focus as the user repeatedly presses the Tab key. This includes

- `SimpleButton` instances
- Input `TextFields`
- `InteractiveObjects` whose `buttonMode` or property is `true`

In addition, any `InteractiveObject` whose `tabEnabled` property is `true` is added to the tab order. Setting `tabEnabled` to `false`, conversely, removes an otherwise tab-able object from the tab order.

This automatic tab ordering doesn't care about a display object's depth, its nesting, or the order it was added. Flash Player tries to make a sensible order out of the tab-enabled objects by ordering them as they are positioned on stage, from left to right and top to bottom. This usually does a passable job, but if it's not cutting the mustard, you can create a fully custom tab order.

Flash Player can operate with either a fully automatic tab order as I've just described, or a fully curated one. The second you start defining your own tab order, Flash Player gives up entirely on automatic tab ordering, and only the objects you've defined tab orders for are reachable by tabbing, so be careful. The transition between the two modes is entirely implicit as well. To return to automatic tab order, you have to make sure that every `InteractiveObject` has an undefined tab order.

To set a custom tab order, use the `tabIndex` property of the `InteractiveObjects` you want to be able to tab to. Assign these properties unique integer indices, starting with 0, in the order you want tabbing to follow. Flash Player treats these indices globally. No matter how deep your objects are or

where they are positioned, they follow these indices unquestioningly. To remove them from the tab order and restore automatic tab ordering, reset all the `tabIndex` properties to their default, −1.

Focus Rectangle

When the keyboard is used to set focus, buttons (that is, `SimpleButton`s and any `InteractiveObject` with `buttonMode` set to `true`) indicate that they have focus by displaying a yellow bounding box around their contents, called the *focus rectangle*.

You can enable or disable the focus rectangle on a case-by-case basis by setting the `focusRect` property of the `InteractiveObject`. Set it to `true`, and a focus rectangle is displayed when tabbed to. Set it to false, and no focus rectangle appears when it is tabbed to. Set it to `null`, however, and it inherits the value from the stage's `stageFocusRect` property.

Because the `focusRect` property of every `InteractiveObject` defaults to `null`, you can disable or enable focus rectangles globally by setting the `stageFocusRect` property on the stage.

If you disable focus rectangles, it's a good idea to provide some other way to indicate that the object is focused. You can create a custom visual treatment and listen to `FocusEvent.FOCUS_IN` and `FocusEvent.FOCUS_OUT` to toggle it.

Context Menus

The secondary mouse button (I'll say *right-clicking* even though your secondary mouse button might not be on the right) triggers a context menu in Flash Player. The context menu is indeed contextual: which menu appears depends on what your mouse is over when you right-click. Just like a mouse event, Flash Player searches for the topmost, visible, filled `InteractiveObject` under the mouse pointer for which mouse events are enabled, to determine the target of your click. If this target has a `contextMenu` property set to an instance of `flash.ui.ContextMenu`, it displays this menu. Otherwise, it travels up the display list to the root display object looking for an `InteractiveObject` with a defined `contextMenu` property. If none is found, Flash Player uses the default context menu for the type of object your pointer is over. There are two kinds of default context menus you can edit: the standard context menu, and a text-editing context menu for all selectable or editable `TextField`s.

Creating and Setting a Context Menu

The most likely place to customize your context menu is at the root `InteractiveObject`. When you create a custom context menu at the root level, it becomes the new *de facto* default context menu, because Flash Player ultimately finds it when searching up the display list for a defined context menu.

Caution

Don't set a context menu on the stage; Flash Player doesn't allow it. Instead, set it on the instance of the Application class in Flash Builder or the Document class in Flash Professional. ∎

Even though right-clicking an uncustomized `InteractiveObject` presents you with a context menu, that doesn't mean it has a `contextMenu` property defined that you can customize — you're

seeing the default menu or parent display object's menu. To set a custom context menu, you have to create a new one:

```
var buttonSprite:Sprite = new Sprite();
var menu:ContextMenu = new ContextMenu();
buttonSprite.contextMenu = menu;
```

Once you create a `ContextMenu` item, you can start customizing the menu by adding and removing menu items.

Customizing Default Items

A new `ContextMenu` contains several built-in menu items, depending on whether the pointer is over an editable `TextField`.

Note
When using context menus in the AIR runtime, there are no default or permanent menu items, and you have fewer limitations and more control over the context menu than in Flash Player. Check the AS3LR for details. ■

Some context menu items are permanent and can't be hidden or removed. Others can be included or hidden by ActionScript. Table 21-3 summarizes all built-in menu items.

You can hide or show these items in detail by toggling Boolean properties of `ContextMenu`'s `builtInItems` property and `clipboardItems` property. Or you can choose to hide all nonpermanent items from a given context menu with a simple method call:

```
var menu:ContextMenu = new ContextMenu();
menu.hideBuiltInItems();
```

After calling this method, all menu items not marked as "Permanent" in Table 21-3 are hidden from the context menu. (Just remember that you have to set that context menu to an `InteractiveObject` before you'll actually see it!)

Adding Custom Items

After cleaning up the menu a bit, you can add up to 15 custom menu items of your own design. These appear above all built-in items.

You add a new context menu item by creating a new instance of `ContextMenuItem`, setting its `caption` property, and adding it to the `customItems` array of the `ContextMenu` you want it to appear in. When the menu item is selected, it broadcasts a `ContextMenuEvent.MENU_ITEM_SELECT` event. Perform the action associated with your new menu item when you receive this event.

You can insert separators into the context menu — usually horizontal rules — by turning on the `separatorBefore` property of the `ContextMenuItem` that should go after the separator.

You can reuse `ContextMenuItems` in multiple menus by cloning them with their `clone()` method. Finally, you can hide them by setting their `visible` property to `false`.

TABLE 21-3

Built-In Context Menu Items

Label	Permanent?	Appears in	Function
Zoom In	No	Normal menu	Zooms in
Zoom Out	No	Normal menu	Zooms out
100%	No	Normal menu	Zooms to normal level
Show All	No	Normal menu	Zooms to show entire stage
Quality	No	Normal menu	Shows quality submenu to change stage rendering quality
Print...	No	Normal menu	Launches Print dialog to print whole contents of stage
Play	No	Multiframe SWFs	Toggles playback of main timeline
Loop	No	Multiframe SWFs	Toggles looping of main timeline
Rewind	No	Multiframe SWFs	Moves the playhead to the beginning of the timeline and first scene
Forward	No	Multiframe SWFs	Moves to the next scene or frame
Back	No	Multiframe SWFs	Moves to the previous scene or frame
Cut	No	TextField menu	Cuts selected text
Copy	No	TextField menu	Copies selected text
Paste	No	TextField menu	Pastes text
Delete	No	TextField menu	Deletes selected text
Select All	No	TextField menu	Selects all text
Show Redraw Regions	Yes	Debug Player	Toggles highlighting of dirty rects
Debugger	Yes	Debug Player	Opens dialog to connect to debugger
Settings...	Yes	Always	Opens settings panel inside Flash Player
About Adobe Flash Player [*version*]...	Yes	Always	Launches About dialog box

The constructor of `ContextMenuItem` optionally lets you set all these properties at once. It has the method signature:

```
function ContextMenuItem(caption:String, separatorBefore:Boolean = false,
                         enabled:Boolean = true, visible:Boolean = true)
```

You must at least set the caption or label of the menu item. The other options are usually correct by default.

457

Let's put all this together in Example 21-13, which revisits the cloning circles from Example 21-6. Instead of cloning by Shift-dragging, I'll add a context menu to all circles with a menu item to create a new one. I'll also customize the default context menu to include an Arrange All option, which will arrange the circles in a grid.

EXAMPLE 21-13   http://actionscriptbible.com/ch21/ex13

Context Menus

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.geom.Point;
  import flash.ui.*;
  [SWF(backgroundColor="0")]
  public class ch21ex13 extends Sprite {
    public function ch21ex13() {
      var circ:DraggableCircle = new DraggableCircle();
      circ.x = circ.y = 100;
      addChild(circ);

      var menu:ContextMenu = new ContextMenu();
      menu.hideBuiltInItems();
      var item:ContextMenuItem;
      item = new ContextMenuItem("Arrange all");
      item.addEventListener(ContextMenuEvent.MENU_ITEM_SELECT, onArrange);
      menu.customItems.push(item);
      contextMenu = menu;
    }
    protected function onArrange(event:ContextMenuEvent):void {
      for (var i:int = 0, x:Number = 50, y:Number = 50; i < numChildren; i++) {
        var circ:DisplayObject = getChildAt(i);
        circ.x = x; circ.y = y;
        if ((x += 50) > stage.stageWidth) {x = 0; y += 50;}
      }
    }
  }
}
import flash.display.*;
import flash.events.*;
import flash.ui.*;
class DraggableCircle extends Sprite {
  public function DraggableCircle() {
    graphics.beginFill(makeColor(), 0.5);
    graphics.drawCircle(0, 0, 50);
    graphics.endFill();
    addEventListener(MouseEvent.MOUSE_DOWN, onStartDrag);
    buttonMode = true;
    blendMode = BlendMode.ADD;

    var menu:ContextMenu = new ContextMenu();
    menu.hideBuiltInItems();
```

```
    var item:ContextMenuItem = new ContextMenuItem("Clone");
    item.addEventListener(ContextMenuEvent.MENU_ITEM_SELECT, onClone);
    menu.customItems.push(item);
    contextMenu = menu;
  }
  protected function makeColor():uint {
    var rnd:Function = function():uint{return uint(Math.random() * 256)};
    return (rnd() << 16 | rnd() << 8 | rnd());
  }
  protected function onStartDrag(event:MouseEvent):void {
    startDrag();
    stage.addEventListener(MouseEvent.MOUSE_UP, onStopDrag);
  }
  protected function onStopDrag(event:MouseEvent):void {
    stage.removeEventListener(MouseEvent.MOUSE_UP, onStopDrag);
    stopDrag();
  }
  public function onClone(event:ContextMenuEvent):void {
    var copy:DraggableCircle = new DraggableCircle();
    copy.x = this.x + 10;
    copy.y = this.y + 10;
    this.parent.addChild(copy);
  }
}
```

In the example, you can see both object-specific context menus and default context menus in action, as well as the stripped-down built-in context menu.

Summary

- ActionScript's event system handles mouse and keyboard events, as instances of `MouseEvent` and `KeyboardEvent`.
- Mouse events set their target to the topmost, visible, nonempty `InteractiveObject` that responds to mouse input.
- Keyboard events set their target to the focused object.
- Both mouse and keyboard events bubble and can be canceled.
- Listen to `stage` to get all mouse or keyboard events regardless of mouse position and focus.
- `mouseEnabled`, `mouseChildren`, `tabEnabled`, and `tabChildren` properties make the display object invisible to or opaque to mouse and focus events.
- Single clicks and double clicks dispatch their own events, although double clicks require `doubleClickEnabled` to be set.
- Mouse events carry information about the position of the cursor with them, as well as any keyboard modifier keys that were held down.

- Use `startDrag()` and `stopDrag()` along with the `MOUSE_DOWN` and `MOUSE_UP` events to click and drag.

- Keyboard input comes in two event types: `KEY_DOWN` and `KEY_UP`.

- `KeyboardEvent` objects carry information about the key that was pressed (`keyCode`) and the character that the key is associated with (`charCode`), as well as information about modifier keys held down.

- Character codes are in UTF-8 and can be converted back to their characters with `String.fromCharCode()`.

- Keyboard input occurs before IMEs convert the input. On some platforms you can control the IME's operation in ActionScript.

- Only one object has focus at a given time. The `Stage` instance keeps a reference to that object in its `focus` property, which can be written to reset focus.

- Changes in focus dispatch `FocusEvent` events. You can prevent focus from changing by canceling some of these objects.

- Pressing Tab on the keyboard cycles through tab-enabled `InteractiveObjects` in the automatic tab order, sorted by their position on-screen.

- You can override automatic tab order by setting `tabIndex` properties on `InteractiveObjects`.

- When the keyboard is used to set focus, a focus rectangle appears around certain display objects unless you set `focusRect` to `false`. Alternatively, you can control focus rectangles globally with the `Stage` instance's `stageFocusRect` property.

- Use tab ordering, focus indication, good labels, and if necessary `AcessibilityProperties` (covered in Chapter 41) to help screen readers access your content.

- Context menus may be customized up to a point by hiding some built-in menu items and adding your own custom ones.

- Assign a `ContextMenu` instance to the `contextMenu` property of the root display object, not the stage, to set the default context menu.

- To respond to a user activating your `ContextMenuItems`, listen for the `ContextMenuEvent` event.

Timers and Time-Driven Programming

The Timer class triggers time-based events in ActionScript 3.0. Nearly any application that involves real-world time uses a timer. Timers can be useful for animation, delaying an action, timing out when an asynchronous action takes too long to execute, triggering an action repeatedly or after a delay, or synchronizing several actions.

ActionScript 3.0 also provides access to several old-school timer methods that aren't object-oriented and don't use the event framework but can have their purposes.

In this chapter, I'll discuss how to set up your own timers, and offer a few tricks to help you get the most out of them.

Timer Basics

For working with time, you'll be using the flash.utils.Timer class.

Each instance of a Timer will, while it's running, repeatedly fire events spaced out by a set amount of time. Using events rather than callback functions not only makes Timers more consistent and easier to use, but it allows you to trigger multiple functions with a single Timer.

Timers are similar in some ways to for loops in that both structures can be used to repeat a bit of code several times but are otherwise quite different. for loops execute every iteration without pause and without coming up for air. No other code is executed until the for loop is finished. Long for loops can prevent your application from drawing the next frame, hanging it. Timers, on the other hand, always space out the execution of their scripts with some type of delay, and other code can execute even if a timer is running.

Creating a Timer

To create a timer, you use the `Timer()` constructor, which takes two arguments:

- `delay` — The number of milliseconds between each time the timer event fires.

- `repeatCount` — The number of times the timer event will be fired. The default value, zero, causes the timer to fire indefinitely until the timer is stopped or the program ends.

Tip

All variables related to delay or time in ActionScript use milliseconds — that is, thousandths of a second. So, the number 1,000 represents one second. If you are working with values in seconds — for example, 120 seconds — it might be easier to read if you write the numbers as `120 * 1000` **instead of** `120000`. ∎

Example 22-1 creates a timer that fires every second and ends after 10 iterations.

EXAMPLE 22-1 `http://actionscriptbible.com/ch22/ex1`

Using a Timer

```
package {
  import com.actionscriptbible.Example;
  import flash.utils.Timer;
  public class ch22ex1 extends Example {
    public static const TIMER_DELAY:Number = 1 * 1000;
    public var timer:Timer;

    public function ch22ex1() {
      timer = new Timer(TIMER_DELAY, 10);
    }
  }
}
```

If you run this program, you'll notice that the timer doesn't do anything yet. That's because you still need to add event listeners to it.

Listening for Timer Events

Every time the timer delay elapses, an event is fired from the timer object. These events use the class `TimerEvent`, a subclass of `Event`. You learned about events in Chapter 20, "Events and the Event Flow." There are two different event types in the `TimerEvent` class:

- `TimerEvent.TIMER` — Dispatched every time the timer delay has elapsed. In other words, it's the ticking of the timer's clock.

- `TimerEvent.TIMER_COMPLETE` — Dispatched when the `repeatCount` is reached. When the `repeatCount` is zero, this event is never dispatched. This is like the "DING!" of an egg timer. When the timer's complete, it stops running.

Example 22-2 adds listeners for both of these events.

EXAMPLE 22-2　`http://actionscriptbible.com/ch22/ex2`

Timers and Event Listeners

```
package {
  import com.actionscriptbible.Example;
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  public class ch22ex2 extends Example {
    public static const TIMER_DELAY:Number = 1 * 1000;
    public var timer:Timer;
    public function ch22ex2() {
      timer = new Timer(TIMER_DELAY, 10);
      timer.addEventListener(TimerEvent.TIMER, onTimer);
      timer.addEventListener(TimerEvent.TIMER_COMPLETE, onTimerComplete);
    }
    protected function onTimer(event:TimerEvent):void {
      trace("Tick.");
    }
    protected function onTimerComplete(event:TimerEvent):void {
      trace("Ding!");
    }
  }
}
```

Running this code still doesn't do anything because you need to start the timer!

Starting, Stopping, and Resetting the Timer

Once your timer is instantiated and your event listeners are added, it's time to start your timer. The `Timer` class offers three methods for controlling the timer:

- `start()` — Starts the timer counting. When the timer is running, the timer's read-only property `running` is set to `true`.

- `stop()` — Stops the timer counting. When the timer is stopped, the `running` property is set to `false`.

- `reset()` — Resets the number of times the timer has repeated since it was started. The `currentCount` property is set back to zero.

Tip

It's easy to forget to start your timer. If you're having trouble, make sure you've called the `start()` method. ∎

Once you add the `start()` method to the program, in Example 22-3, things start happening.

EXAMPLE 22-3   http://actionscriptbible.com/ch22/ex3

Starting the Timer

```
package {
  import com.actionscriptbible.Example;
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  public class ch22ex3 extends Example {
    public static const TIMER_DELAY:Number = 1 * 1000;
    public var timer:Timer;
    public function ch22ex3() {
      timer = new Timer(TIMER_DELAY, 10);
      timer.addEventListener(TimerEvent.TIMER, onTimer);
      timer.addEventListener(TimerEvent.TIMER_COMPLETE, onTimerComplete);
      timer.start();
    }
    protected function onTimer(event:TimerEvent):void {
      trace("Tick.");
    }
    protected function onTimerComplete(event:TimerEvent):void {
      trace("Ding!");
    }
  }
}
```

Running this program should show the following output over the course of 10 seconds.

```
Tick.
Tick.
Tick.
Tick.
Tick.
Tick.
Tick.
Tick.
Tick.
Tick.
Ding!
```

Handling Timer Events

Now that you've looked at how to set up a timer and listen to the events it dispatches, let's look at some practical ways to use these events.

The `Timer` class fires `TimerEvent` event objects. These events are ultimately not much different from the base `Event` class. No additional information is stored within a `TimerEvent` that's not in a

regular `Event`. You can get information about the `Timer` that fired the event using the event object's `target` property.

With the `Timer` reference, you can get more information such as the number of times the timer has fired or the time between each tick, as shown in Example 22-4.

EXAMPLE 22-4   `http://actionscriptbible.com/ch22/ex4`

Getting the Timer from the Event Object

```
package {
  import com.actionscriptbible.Example;
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  public class ch22ex4 extends Example {
    public static const TIMER_DELAY:Number = 0.5 * 1000;
    public var timer:Timer;
    public function ch22ex4() {
      timer = new Timer(TIMER_DELAY, 10);
      timer.addEventListener(TimerEvent.TIMER, onTimer);
      timer.addEventListener(TimerEvent.TIMER_COMPLETE, onTimerComplete);
      timer.start();
    }
    protected function onTimer(event:TimerEvent):void {
      var timer:Timer = Timer(event.target);
      var timeElapsed:Number = timer.currentCount * timer.delay;
      var remainingCount:Number = timer.repeatCount - timer.currentCount;
      trace("Time elapsed :", timeElapsed / 1000, "seconds.");
      if (remainingCount > 0) {
        trace("There are", remainingCount, "ticks remaining.");
      }
    }
    protected function onTimerComplete(event:TimerEvent):void {
      trace("Ding!");
    }
  }
}
```

The event handler `onTimer()` prints out the time elapsed in seconds every time the `TIMER` event is fired. The output of Example 22-4 should look something like this:

```
...
There are 3 ticks remaining.
Time elapsed : 4 seconds.
There are 2 ticks remaining.
Time elapsed : 4.5 seconds.
There are 1 ticks remaining.
Time elapsed : 5 seconds.
Ding!
```

In the example, you useds three properties of the `Timer` class:

- `currentCount` — The number of times the timer has fired since it was started. If you use the `reset()` command, this number is reset to 0. This is a read-only property, so it can't be changed directly.

- `repeatCount` — The number of times the timer event will be fired. The default value, zero, causes the timer to fire indefinitely until the timer is stopped or the program ends. This is the same value that is defined when you create the timer instance. `repeatCount` is a read/write property, so it can be changed on the fly.

- `delay` — The number of milliseconds between each firing of the timer event. This is the same value that is defined when you create the timer instance. `delay` is a read/write property, so it can be changed on the fly.

Delaying the Execution of a Function

Timers make it easy to delay the execution of a function. Simply set up a timer that fires only once, and set the delay for the time you'd like to wait before firing. A common application for this is creating a *timeout*, a predetermined time allotted for an action to take place, such as a server response, after which the action is aborted.

In Example 22-5, you'll use a timer to create something you're no doubt familiar with from browsing the internet, a redirect script. First you'll create the `Redirect` class, which will handle the timer and navigation. Then you'll set up an example class to test it.

EXAMPLE 22-5 `http://actionscriptbible.com/ch22/ex5`

Delaying a Function's Execution

```
package {
  import com.actionscriptbible.Example;
  import flash.text.*;
  public class ch22ex5 extends Example {
    protected const redirectURL:String = "http://www.wonderfl.net";
    protected const redirectDelay:Number = 5 * 1000;
    public function ch22ex5() {
      var redirect:Redirect = new Redirect(redirectURL, redirectDelay);

      trace("This page no longer exists. You will be redirected to\n"
            + redirectURL + " after " + redirectDelay/1000 + " seconds.");
    }
  }
}
import flash.utils.Timer;
import flash.events.*;
import flash.net.*;
class Redirect {
  private var redirectTimer:Timer;
  private var redirectURL:URLRequest;
  private static const REDIRECT_DELAY:int = 5000;
```

```
  public function Redirect(url:String, delay:Number=REDIRECT_DELAY) {
    redirectURL = new URLRequest(url);
    redirectTimer = new Timer(delay, 1);
    redirectTimer.addEventListener(TimerEvent.TIMER,onRedirect);
    redirectTimer.start();
  }
  private function onRedirect(event:Event):void {
    navigateToURL(redirectURL, "_self");
  }
}
```

The result should be a page with the redirect message that navigates away after 5 seconds. You'll have to run this one on your own computer, because the security policy where the example is hosted won't let you navigate away.

Tip

You can create a simple timeout with the function `flash.utils.setTimeout()`, covered in the section "Other Time Related Functions." ∎

Creating a World Clock

Let's try a quick and easy example that uses the `Timer` class's ability to send timing events to multiple recipients. Example 22-6 shows a simple world clock that shows the time in multiple time zones. This leverages what you learned about time zones and the `Date` class in Chapter 7, "Numbers, Math, and Dates."

EXAMPLE 22-6 `http://actionscriptbible.com/ch22/ex6`

A World Clock

```
package {
  import flash.display.Sprite;
  import flash.utils.Timer;
  public class ch22ex6 extends Sprite {
    public function ch22ex6()
    {
      var timer:Timer = new Timer(1000);
      var nyc:LocalClock = new LocalClock(timer, "New York City, USA", -5);
      var paris:LocalClock = new LocalClock(timer, "Paris, France", 1);
      var tokyo:LocalClock = new LocalClock(timer, "Tokyo, Japan", 10);

      addChild(nyc);
      nyc.x = 0;
      addChild(paris);
```

continued

EXAMPLE 22-6   *(continued)*

```
        paris.y = 50;
        addChild(tokyo);
        tokyo.y = 100;
        timer.start();
      }
    }
  }
import flash.utils.Timer;
import flash.events.*;
import flash.display.Sprite;
import flash.text.*;
class LocalClock extends Sprite {
  private var location:String;
  private var timezoneOffset:int;
  private var labelTF:TextField;
  private var clockTF:TextField;
  public function LocalClock(timer:Timer, location:String, tzOffset:int) {
    this.location = location;
    this.timezoneOffset = tzOffset;
    labelTF = new TextField();
    clockTF = new TextField();
    labelTF.autoSize = clockTF.autoSize = TextFieldAutoSize.LEFT;
    labelTF.width = labelTF.height = clockTF.width = clockTF.height = 0;
    labelTF.selectable = clockTF.selectable = false;
    labelTF.defaultTextFormat = new TextFormat("_serif", 12, 0, false, true);
    clockTF.defaultTextFormat = new TextFormat("_typewriter", 12, 0x6AF685);
    clockTF.background = true;
    clockTF.backgroundColor = 0x000000;
    labelTF.text = location;
    clockTF.y = labelTF.textHeight + 5;
    addChild(labelTF);
    addChild(clockTF);
    timer.addEventListener(TimerEvent.TIMER, onTimer);
  }
  private function onTimer(event:TimerEvent = null):void {
    var date:Date = new Date();
    date.hoursUTC += timezoneOffset;
    clockTF.text = pad(date.hoursUTC) + ":"
                + pad(date.minutesUTC) + ":"
                + pad(date.secondsUTC);
  }
  private function pad(n:Number):String {
    var s:String = n.toString();
    while (s.length < 2) s = "0" + s;
    return s;
  }
}
```

Each clock is set up with a timer so that it can subscribe to the tick events. These happen to be dispatched once a second. The location and time zone offset are saved and used to create the readout. Each time the `TIMER` event fires, the date is updated and the time zone offset is applied. Then the local time is displayed in the readout.

In the example code, you can edit the time zones to show whatever cities are most useful for you, or you can add an arbitrary number of clocks. Notice that all three clocks are able to use the same `Timer` instance.

Enterframe Events

I touched on enter frame events in Chapter 14, "Visual Programming with the Display List." Display objects that are on the display list dispatch two events — `Event.ENTER_FRAME` and `Event.EXIT_FRAME` — as frames are drawn, one before the frame is drawn, and one after. The rate that frames are drawn is set by the stage's frame rate (which you can modify in the `frameRate` property), but it is not guaranteed. Nonetheless, triggering updates from frame events is a good idea because it ensures that your updates are performed at the same rate as the display is refreshed. You have seen this used in numerous examples by now, such as Example 7-1 (http://actionscriptbible.com/ch7/ex1). I'll introduce more information about using enterframe events for animation in Chapter 39, "Scripting Animation."

Other Time-Related Functions

Although I recommend in general that you use Timers, sometimes it's more convenient and simpler to use one of the procedural time functions. These functions are included in the `flash.utils` package along with `Timer`.

getTimer()

The `getTimer()` function — which, incidentally, has nothing to do with the `Timer` class — returns the number of milliseconds that have elapsed since the program was initialized. `getTimer()` can be a useful tool when you simply need to measure the duration of an action, such as an external XML file loading, without dealing with the overhead of the `Timer` class.

To find an elapsed time, you can store one value from `getTimer()` as your starting time and subtract it from another time when the action is finished:

```
var startTime:Number = getTimer();
// Do some action that might take a while
var endTime:Number = getTimer();
var elapsedTime:Number = endTime - startTime;
```

The current time is returned when you use `getTimer()`, regardless of whether a new frame has been rendered. This is great because a `Timer` can actually run behind the requested interval or be slightly off. `getTimer()` allows you to correct for these differences.

setTimeout()

The `setTimeout()` function allows you to execute a function once after a predefined delay. It doesn't create objects or require listening to events, so it can be called in one line. It has some arguments, so let's look at how to call it.

```
function setTimeout(closure:Function, delay:Number, ...arguments):uint
```

The `closure` is any function. You can point it to a method in the current class, a method on another instance, a variable of type `Function`, or type a function inline. The `delay` is in milliseconds. You can also follow these with any number of arguments that will be passed to the `closure` after the delay. It returns a unique ID that can be used to cancel the call with `clearInterval()`. This can be nice to use occasionally because it doesn't have as much code overhead as a `Timer`, and the function you trigger doesn't have to be an event listener (whose signature, remember, must have only one argument for the event object).

setInterval()

The `setInterval()` function is almost identical to `setTimeout()`, except that it keeps calling the closure at intervals of `delay`. It can only be cancelled by a call to `clearInterval()`. I would steer clear of this method and use `Timer`s if you need to do something repeatedly, because it's kind of crummy to keep around numeric IDs to cancel `setInterval()`. It's much more object oriented to associate an object — a `Timer` — with the timed actions.

clearInterval()

Call `clearInterval()` with the ID returned from a `setInterval()` or `setTimeout()` call to cancel that interval. The delayed action is canceled immediately. Again, managing these global interval IDs is very un-OOP, so if you find yourself needing `clearInterval()`, it's time to think about using a `Timer`.

Aside: Threads

One thing that's sorely missing from ActionScript 3.0 is the ability to execute code *asynchronously*. Flash Player executes all ActionScript code associated with a frame sequentially before drawing the next frame. This means that operations that take a long time can appear to hang your program, because the heavy workload prevents user interaction from being processed or new frames from being drawn. You'll see this fairly frequently during initialization of a site. In many cases an entire site is built and initialized after the first frame renders, leading to a slowdown during initialization.

You can use timers and enterframe events, however, to fake asynchronous code execution, or threads. If you can break your code down into units that can execute sequentially, a thread manager can execute your task bit by bit, while preventing overall slowdown. It can do this by measuring time that actually elapses between frames with `getTimer()` and comparing it to the target frame rate of the SWF. By executing fewer chunks of an overall task, the thread manager can reduce time spent on any given frame, keeping things smooth.

It should be noted that what's described are not real threads, but merely an ActionScript-based runtime simulation of threading. Sometimes this is referred to as *green threading*. One implementation

is the Spark Project's thread library, Soumen (`http://www.libspark.org/wiki/Thread/en`).
Additionally, Drew Cummins wrote up a framework and includes demos and source
(`http://bit.ly/cummins-threads`).

Summary

- The `Timer` class is an object oriented way to manage timed actions in ActionScript 3.0.
- `Timers` are useful for a number of things including delaying a function, repeating a function, and synchronizing several functions.
- `Timers` have a delay time and a repeat count. Setting the repeat count to zero causes them to continue firing indefinitely.
- `Timers` fire the events `TimerEvent.TIMER` and `TimerEvent.TIMER_COMPLETE`.
- `Event.ENTER_FRAME` is dispatched by display objects on the display list before each frame is rendered. This event can be used for animation.
- The `getTimer()` function retrieves the time in milliseconds since the application started up and is useful for accurate time measurements.

Multitouch and Accelerometer Input

Due to the efforts of the Open Screen Project (`http://openscreen` `project.org`), Flash Player is appearing on a growing number of smartphones. Simultaneously, consumers, desktop and laptop manufacturers, and makers of desktop operating systems are embracing touchscreens. The net effect is that Flash Player has access to a wider array of input than ever before. The user is no longer limited to a keyboard and mouse. In fact, she may not even have a keyboard or mouse installed, although chances are in that case that a software keyboard is provided and touch input can be interpreted as mouse input.

In this chapter you'll learn how to deal with not just touch-based input, but information from various sensors Flash Player may have access to, including accelerometers, and geolocation sensors such as GPS. I'll mention it, but I won't go into depth on geolocation because it's currently available only to AIR or native apps. Between the keyboard and mouse (Chapter 21, "Interactivity with the Mouse and Keyboard"), the camera and microphone (Chapter 33, "Capturing Sound and Video"), and touch and accelerometer (this chapter), modern hardware has a surprisingly extensive repertoire of interaction modes.

FEATURED CLASSES

`flash.ui.Multitouch`

`flash.events.TouchEvent`

`flash.events`
  `.GestureEvent`

`flash.sensors`
  `.Accelerometer`

`flash.events`
  `.AccelerometerEvent`

Planning for Your Audience

Consider three platforms you can find Flash Player on: a phone with a 320 × 480 pixel touch display and keyboard; a 720 × 1280 pixel TV with no mouse or touch, only a remote; and a desktop PC with a 2560 × 2048 pixel display, mouse, and keyboard. The users of these different platforms can't be expected to interact with them in the same way. When you're writing an ActionScript program, then, you must consider your audience and plan appropriately.

This doesn't mean that you have to redo your application for every possible combination of hardware. You can limit yourself to a specific audience. You can develop a game specifically for a multitouch capable small-screen device; it won't work right on a desktop with a mouse, and that's fine. It shouldn't.

On the other hand, you may need to ensure your application is available as widely as possible. In this case, you may find it easier to create more than one version of the application, serving the correct one to the correct user. Or, you can detect certain features and properties of the device Flash Player is running on and change the way your program behaves at runtime to better suit each user's specific needs. In most cases, however, the difference in processing power and screen size requires you to rethink or redesign an application for a smartphone versus a desktop PC.

I only mention this because thinking about it before you start programming is essential. With Flash Player on so many different kinds of devices, you can't assume the user will have a big screen, fast CPU, mouse, and keyboard, or a multitouch input device for that matter.

Note
Hypocrisy alert: many of the examples in this book don't check whether you have an appropriate device or tailor themselves for a certain size of screen. Remember, example code is a far cry from production-grade software. (Sorry!) ∎

Using Multitouch

In case you missed the outrageous hype, a *multitouch* input device is one that can detect and track more than one unique pointer at the same time. Most of the time, that pointer is a finger, and the device is a screen. I can use a stylus on my multitouch screen, and Mac laptops have multitouch trackpads rather than screens, but for convenience's sake, I'll write as if you're touching a screen with your fingers.

The *raison d'être* of multitouch displays is to make interfaces more "natural" to use. So, common uses of multitouch include swiping to progress through pages, pinching to zoom, twisting to rotate, and so on. You can use these commonly recognized touch *gestures*, or you can interpret touches in your own way. If used properly, multitouch input can make an interface a joy to use or enable new kinds of applications and games that would not be possible without it.

Flash Player 10.1 and later support multitouch screens. To use multitouch, you need to have appropriate hardware, drivers, and operating system support. At the time of this writing, Flash Player support for multitouch was limited to Windows 7 and iPhone OS, with some support on Mac OS X 10.5.3 and later (`http://bit.ly/fp-beta-multitouch-support`). There's a chance this has changed, so check Adobe's Flash Player release notes to see which platforms can support multitouch in Flash Player.

Version
FP10.1. Multitouch input is available only in Flash Player 10.1 and above; a later version may be needed to support additional devices or operating systems. ∎

Because support for multitouch is so varied, you should check on the host's multitouch capabilities before you try to use any. Multitouch setup properties are stored as static properties of the `Multitouch` class in `flash.ui`.

First of all, this class was added in Flash Player 10.1. Earlier versions of Flash Player can still run code targeted for Flash Player 10.1; so accessing the `Multitouch` class causes older versions of Flash Player to throw an error. If you don't have the `Multitouch` class, you certainly don't have access to Flash Player's multitouch API, so this is the first thing you should check. As soon as you find out that `Multitouch` doesn't exist, make sure to skip any and all code that uses it.

```
try {
  var test:Class = Multitouch;
  //anything using the Multitouch class
} catch (error:ReferenceError) {
  //you must not have FP10.1+
  trace("ERROR: Multitouch not supported in your runtime.");
}
```

Caution

You may be tempted to test against a particular Flash Player version number in your code. Although that may work, it isn't as maintainable, and it isn't as self-explanatory, as catching the `ReferenceError.` ∎

If you're all set and the `Multitouch` class is there, you can use these static properties to find out more about the environment:

- `supportsGestureEvents` — `true` if the hardware and OS can recognize common multi-touch gestures
- `supportedGestures` — A `Vector` of gesture names (`Strings`) that the hardware and OS can recognize
- `supportsTouchEvents` — `true` if the hardware and OS can send the raw touch events
- `maxTouchPoints` — The number of simultaneous touches the hardware can recognize (an `int`)

You can make sure that a multitouch device is available by checking if (`Multitouch .maxTouchPoints > 1`). In Example 23-1, you'll display the properties of a multitouch display, remember whether multitouch is available, and prioritize one mode over another, all using version-sensitive code that won't throw an error.

EXAMPLE 23-1 `http://actionscriptbible.com/ch23/ex1`

Retrieving Multitouch Capabilities

```
package {
  import com.actionscriptbible.Example;
  import flash.system.Capabilities;
  import flash.ui.Multitouch;
  import flash.ui.MultitouchInputMode;
  public class ch23ex1 extends Example {
    protected var isMultitouch:Boolean;
    public function ch23ex1() {
      try {
        var test:Class = Multitouch; //older FP should throw error here
        trace("Multitouch capabilities on this device:");
        trace("Screen type:", Capabilities.touchscreenType);
        trace("Touch-level access?", Multitouch.supportsTouchEvents);
        trace("Gesture-level access?", Multitouch.supportsGestureEvents);
        trace("Number of touch points:", Multitouch.maxTouchPoints);
        if (Multitouch.supportsGestureEvents) {
          trace("supported gestures {");
          for each (var gestureName:String in Multitouch.supportedGestures) {
            trace("  -", gestureName);
```

EXAMPLE 23-1 *(continued)*

```
      }
      trace("}");
    }
    //remember whether multitouch mode is on
    isMultitouch = Multitouch.maxTouchPoints > 1;
    //prefer gesture mode if available
    if (Multitouch.supportsGestureEvents) {
      Multitouch.inputMode = MultitouchInputMode.GESTURE;
    } else if (Multitouch.supportsTouchEvents) {
      Multitouch.inputMode = MultitouchInputMode.TOUCH_POINT;
    }
  } catch (error:ReferenceError) {
    trace("Sorry, but multitouch is not supported in this runtime.");
    isMultitouch = false;
  }
 }
 }
}
```

This is a good starting point for boilerplate code that toggles multitouch features based on availability.

In this example, you also use `Multitouch.inputMode`, which tells Flash Player which mode it should engage the multitouch display in. This is the only of the `Multitouch` properties that can be set as well as read. Besides the two options in the example, there's a `MultitouchInputMode.NONE` mode, which ignores the special nature of touch and converts touch events into mouse events when possible.

The two multitouch modes — touch mode and gesture mode — behave very differently. Gesture mode lets the OS interpret touches into gestures, while touch mode lets you watch every action of every finger. Unfortunately, it's all-or-nothing. You have to choose the correct mode for the job (although don't forget that you can switch modes at runtime).

Touch Mode

Touch mode is the more powerful of the two modes. In touch mode, you can see every motion of every finger, almost like having multiple mice. Touch events, represented by the `TouchEvent` class, are broadcast by the innermost and topmost `InteractiveObject` under the touch location, and they bubble up to the stage, just like `MouseEvents`.

To use touch mode, just set

```
Multitouch.inputMode = MultitouchInputMode.TOUCH_POINT;
```

And add event listeners normally. You can track these types of events related to each touch:

- `TouchEvent.TOUCH_BEGIN` — Like `MouseEvent.MOUSE_DOWN`
- `TouchEvent.TOUCH_END` — Like `MouseEvent.MOUSE_UP`
- `TouchEvent.TOUCH_MOVE` — Like `MouseEvent.MOUSE_MOVE` while the mouse button is down

- `TouchEvent.TOUCH_OVER` — Like `MouseEvent.MOUSE_OVER` while the mouse button is down
- `TouchEvent.TOUCH_OUT` — Like `MouseEvent.MOUSE_OUT` while the mouse button is down
- `TouchEvent.ROLL_OVER`
- `TouchEvent.ROLL_OUT`
- `TouchEvent.TOUCH_TAP` — Like `MouseEvent.CLICK`

The difference between these touch events and their corresponding mouse events is that a multitouch device only registers a touch while your finger is down. So `TOUCH_MOVE` is more like what you'd think of as a mouse drag. Although it might be my hardware or a bug, I find `ROLL_OVER` and `ROLL_OUT` aren't fired consistently; I use `TOUCH_OVER` and `TOUCH_OUT` instead.

Event Object Properties Shared by MouseEvent

Like `MouseEvent`, `TouchEvent` objects are associated with an `InteractiveObject target`, and as they bubble, their `currentTarget` changes. Their position, the position of your finger, is exposed relative to the `target` as `localX` and `localY`, or in global coordinates as `stageX` and `stageY`.

Also like `MouseEvent`, although less likely to be used, `TouchEvents` let you know the state of keyboard modifiers, such as `shiftKey`, `ctrlKey`, and `altKey`.

Event Object Properties Unique to TouchEvent

The most important addition to `TouchEvent` objects is the `touchPointID` property. If multiple fingers are down, each finger may be triggering `TouchEvents` simultaneously. These touches would be impossible to distinguish, except that Flash Player assigns each continuous touch a unique `touchPointID`.

The ID is assigned when your finger first touches the screen. Anything that your finger does while it's pressed against the screen generates `TouchEvents` with that `touchPointID`. And finally, you can forget all about the ID as soon as you lift your finger. The system has no clue which finger you're using: if you use your pinky one time and your ring finger the next, or your index finger both times, it could care less. But it does know that whatever fleshy appendage A smashed into the screen is the same appendage A that rolled over button B a second later. The system may recycle the same numbers for these IDs, but there's no continuity between `touchPointIDs` after the finger is lifted.

When you're tracking multiple concurrent touches, you might want to store the current state of each touch in an array by its `touchPointID` so that you can easily update the correct one every time you get a `TouchEvent`.

If you want to ignore every finger that comes after the first in a multifinger gesture, or if you want to treat the first finger pressed in a special way, there's a simpler way. `TouchEvents` that are triggered by the so-called primary finger have their `isPrimaryTouchPoint` property set to `true`.

If the hardware and OS support these properties, you may also be able to find out more about the touch with these `TouchEvent` properties:

- `sizeX`, `sizeY` — The size of the contact area between your finger and the screen. Relative size, not measured in pixels (at least on my hardware).
- `pressure` — How hard your finger is pressing down, between 0.0 (are you even trying?) and 1.0 (don't break your monitor, dude).

These properties can bring a whole new level of detail to your multitouch application.

Putting It Together

Something fun like multitouch calls for a fun example, don't you think? I saw this little heat-activated mood toy on the iPhone app store and thought it would make a great example (Example 23-2). It does, however, use some graphics techniques from Chapters 34 to 37, so this is a taste of what you can do in a short space with interactive graphics and multitouch.

EXAMPLE 23-2   http://actionscriptbible.com/ch23/ex2

Multitouch in Touch Mode

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.filters.BlurFilter;
  import flash.geom.*;
  import flash.ui.*;
  public class ch23ex2 extends Sprite {
    protected const BMP_SCALE:Number = 1/2;
    protected const D:Number = 1.015;
    protected const DIM_EFFECT:ColorTransform = new ColorTransform(D, D, D);
    protected const B:int = 16;
    protected const BLUR_EFFECT:BlurFilter = new BlurFilter(B, B, 1);
    protected var RLUT:Array, GLUT:Array, BLUT:Array;
    protected var sourceBmp:BitmapData;
    protected var colorBmp:BitmapData;
    protected var touches:Array = new Array();
    protected var fingerShape:Shape = new Shape();
    public function ch23ex2() {
      try {
        var test:Class = Multitouch;
        if (Multitouch.supportsTouchEvents) {
          Multitouch.inputMode = MultitouchInputMode.TOUCH_POINT;
          init();
        } else {
          trace("Sorry, this example requires multitouch.");
        }
      } catch (error:ReferenceError) {
        trace("Sorry, but multitouch is not supported in this runtime.");
      }
    }
    protected function init():void {
      //create a black-and-white bitmap and a color bitmap, only show the color
      sourceBmp = new BitmapData(
        stage.stageWidth*BMP_SCALE, stage.stageHeight*BMP_SCALE, false, 0);
      colorBmp = sourceBmp.clone();
      var bitmap:Bitmap = new Bitmap(colorBmp, PixelSnapping.ALWAYS, true);
      bitmap.width = stage.stageWidth; bitmap.height = stage.stageHeight;
      addChild(bitmap);

      //create finger shape to paste onto the bitmap under your touches
      fingerShape.graphics.beginFill(0xffffff, 0.1);
      fingerShape.graphics.drawEllipse(-15, -20, 30, 40);
      fingerShape.graphics.endFill();
```

```
    //create the palette map from a gradient
    var gradient:Shape = new Shape();
    var m:Matrix = new Matrix();
    m.createGradientBox(256, 10);
    gradient.graphics.beginGradientFill(GradientType.LINEAR,
      [0x313ad8, 0x2dce4a, 0xdae234, 0x7a1c1c, 0x0f0303],
      [1, 1, 1, 1, 1], [0, 0.4*256, 0.75*256, 0.9*256, 255], m);
    gradient.graphics.drawRect(0, 0, 256, 10);
    var gradientBmp:BitmapData = new BitmapData(256, 10, false, 0);
    gradientBmp.draw(gradient);
    RLUT = new Array(); GLUT = new Array(); BLUT = new Array();
    for (var i:int = 0; i < 256; i++) {
      var pixelColor:uint = gradientBmp.getPixel(i, 0);
      //I drew the gradient backward, so sue me
      RLUT[256-i] = pixelColor & 0xff0000;
      GLUT[256-i] = pixelColor & 0x00ff00;
      BLUT[256-i] = pixelColor & 0x0000ff;
    }

    stage.addEventListener(TouchEvent.TOUCH_BEGIN, assignTouch);
    stage.addEventListener(TouchEvent.TOUCH_MOVE, assignTouch);
    stage.addEventListener(TouchEvent.TOUCH_END, removeTouch);
    stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  protected function assignTouch(event:TouchEvent):void {
    touches[event.touchPointID] = event;
  }
  protected function removeTouch(event:TouchEvent):void {
    delete touches[event.touchPointID];
  }
  protected function onEnterFrame(event:Event):void {
    for (var key:String in touches) {
      var touch:TouchEvent = touches[key] as TouchEvent;
      if (touch) {
        //plaster the finger image under your finger
        var m:Matrix = new Matrix();
        m.translate(touch.stageX*BMP_SCALE, touch.stageY*BMP_SCALE);
        sourceBmp.draw(fingerShape, m, null, BlendMode.ADD);
      }
    }
    var O:Point = new Point(0, 0);
    //blur and ever-so-slightly brighten the image to make the color last
    sourceBmp.applyFilter(sourceBmp, sourceBmp.rect, O, BLUR_EFFECT);
    sourceBmp.colorTransform(sourceBmp.rect, DIM_EFFECT);
    //you've calculated the image in grayscale brightnesses, now make it color
    colorBmp.paletteMap(sourceBmp, sourceBmp.rect, O, RLUT, GLUT, BLUT, null);
  }
}
}
```

In this example, you want the position of the current touches every frame, even if they're not moving, so you can hold down your finger and gradually add to the "heat" of the surrounding area. However, you only get the finger location from a `TouchEvent` object, unlike the mouse location, which you can get from any display object. To get around this, I've stored the most recent event objects by their `touchPointID`, removing them when the finger is lifted. Then the location can be retrieved from the event object at any time.

A modified version of this example that allows you to use the mouse if you don't have multitouch is included online. This kind of fallback behavior is critical to accommodate as many users as possible.

Gesture Mode

In gesture mode, the host operating system interprets the raw touch data for you, into a vocabulary of gestures. This has clear pros and cons:

- PRO — You don't have to come up with algorithms that interpret touches as gestures.
- PRO — Your application uses a gesture vocabulary consistent with the rest of the host OS.
- CON — You don't have access to the raw touch events any more.
- CON — You're limited to the gestures that the host OS provides and the way it interprets touches into gestures (which can be error-prone).
- CON — The user can't combine gestures; only one gesture at a time is recognized.

As you've already seen, you can determine if gestures are supported by querying `Multitouch.supportsGestureEvents`. If they are, switch to gesture mode by setting

```
Multitouch.inputMode = MultitouchInputMode.GESTURE;
```

If you require a specific gesture to be supported, you can check against the `supportedGestures` list that you learned about in the beginning of the section.

Table 23-1 summarizes all the gestures you can listen for.

Each gesture event object has a `phase` property that tells you if the gesture is beginning, ending, or if it's underway and it's changing (for example, as a pan gesture pans around). These phases are the string constants `GesturePhase.BEGIN`, `GesturePhase.END`, and `GesturePhase.UPDATE`, in the `flash.events` package. It's possible for an event to begin but never end (a two-finger tap should be like this), or end without updating.

In gesture mode, the OS interprets normal, single-finger taps as mouse events; you can listen for them with the normal mouse events.

As in touch mode, these events are assigned an `InteractiveObject` target and bubble. They have the same modifier key properties and the same `localX`, `localY`, `stageX`, and `stageY` properties.

Special Properties of PressAndTapGestureEvent

The `PressAndTapGestureEvent` object contains one additional set of properties, which locate the position that the second finger tapped. The primary location of the event is where you placed your first finger. The properties `tapLocalX`, `tapLocalY`, `tapStageX`, and `tapStageY` identify where the second finger tapped, relative to `target` and `stage`, respectively, as figure 23-1 illustrates.

Gesture Events

Event Class	Event Name	Gesture
GestureEvent	GESTURE_TWO_FINGER_TAP	Tap quickly with two fingers close to each other.
PressAndTapGestureEvent	GESTURE_PRESS_AND_TAP	Hold down one finger and tap quickly with the other. You can keep moving the first finger after lifting the second finger.
TransformGestureEvent	GESTURE_PAN	Place two fingers and move them slowly in unison.
TransformGestureEvent	GESTURE_ROTATE	Place two fingers and rotate them around their center point.
TransformGestureEvent	GESTURE_SWIPE	Place two fingers and whisk them in a direction, leaving the surface.
TransformGestureEvent	GESTURE_ZOOM	Place two fingers and spread them apart or pinch them together around their center point.

FIGURE 23-1

The press-and-tap gesture and the positions of the two fingers

(tapLocalX, tapLocalY)

(localX, localY)

Special Properties of TransformGestureEvent

You can use different transformation gestures to apply affine transformations in a natural way. When you use `TransformGestureEvents` — pan, rotate, swipe, and zoom — the host environment figures out how to translate the finger movements into these transformations. It stores these in the following properties of the event object:

- `offsetX`, `offsetY` — The translation since the last event in this gesture
- `rotation` — The rotation (in degrees) since the last event in this gesture
- `scaleX`, `scaleY` — The scale since the last event in this gesture

Putting It Together

You can use the fact that there are a variety of different gestures to manipulate an object in three dimensions. In Example 23-3, you'll use a two-finger pan gesture in 2D to rotate a plane around the x- and y-axes, a rotation gesture to rotate it around the z-axis, a pinch to scale it up and down, and a two-finger tap to reset it. This feels slightly more natural than trying to control all three dimensions with a 2D mouse cursor.

EXAMPLE 23-3   http://actionscriptbible.com/ch23/ex3

Multitouch in Gesture Mode

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.geom.Point;
  import flash.geom.Rectangle;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  import flash.ui.Multitouch;
  import flash.ui.MultitouchInputMode;

  [SWF(backgroundColor="#404040", width="1024", height="768")]
  public class ch23ex3 extends Sprite {
    protected var holder:DisplayObjectContainer;
    protected var cropRect:Rectangle;
    public function ch23ex3() {
      try {
        var test:Class = Multitouch;
        if (Multitouch.supportsGestureEvents) {
          Multitouch.inputMode = MultitouchInputMode.GESTURE;
          var l:Loader = new Loader();
          l.load(new URLRequest(
            "http://actionscriptbible.com/files/bluemarble.jpg"),
            new LoaderContext(true)
          );
          l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoadComplete);
        } else {
          trace("Sorry, this example requires multitouch.");
        }
      } catch (error:ReferenceError) {
```

```
      trace("Sorry, but multitouch is not supported in this runtime.");
    }
  }
  protected function onLoadComplete(event:Event):void {
    this.x = stage.stageWidth/2; this.y = stage.stageHeight/2;
    this.transform.perspectiveProjection.projectionCenter =
      new Point(stage.stageWidth/2, stage.stageHeight * 0.25);
    this.transform.perspectiveProjection.fieldOfView = 70;

    var content:DisplayObject = LoaderInfo(event.target).content;
    holder = LoaderInfo(event.target).loader;
    content.x = -content.width/2; content.y = -content.height/2;
    var targetWidth:Number = stage.stageWidth * 0.7;
    var scale:Number = Math.min(1, targetWidth / content.width);
    holder.z = 100;
    holder.scaleX = holder.scaleY = scale;
    addChild(holder);
    stage.addEventListener(GestureEvent.GESTURE_TWO_FINGER_TAP, onDoubleTap);
    stage.addEventListener(TransformGestureEvent.GESTURE_PAN, onGesture);
    stage.addEventListener(TransformGestureEvent.GESTURE_ROTATE, onGesture);
    stage.addEventListener(TransformGestureEvent.GESTURE_SWIPE, onGesture);
    stage.addEventListener(TransformGestureEvent.GESTURE_ZOOM, onGesture);
  }
  protected function onDoubleTap(event:GestureEvent):void {
    holder.transform.matrix3D.identity();
  }
  protected function onGesture(event:TransformGestureEvent):void {
    switch (event.type) {
      case TransformGestureEvent.GESTURE_PAN:
        var SCALE:Number = 0.5;
        holder.rotationY -= event.offsetX * SCALE;
        holder.rotationX += event.offsetY * SCALE;
        break;
      case TransformGestureEvent.GESTURE_ROTATE:
        holder.rotationZ += event.rotation;
        break;
      case TransformGestureEvent.GESTURE_ZOOM:
        holder.scaleX *= event.scaleX;
        holder.scaleY = holder.scaleX;
        break;
    }
  }
}
```

There's a lot you can do with these built-in gestures. Hopefully when you build your applications they will be more mouse-friendly and more obvious what to do than this example. Unlike a menu or a button, which contains a pithy self-description, touch interactions are not visible, so you have to make them second-nature, explain them, or make them optional.

Touch-Related Methods

In addition to the events added for touch, `Sprite` two methods to enable touch-driven drag-and-drop. Called `startTouchDrag()` and `stopTouchDrag()`, these have the same interface and usage as `startDrag()` and `stopDrag()`, but with the addition of an argument for the `touchPointID`. This way, you can have multiple simultaneous drag events, each with its own touch.

In Example 23-4, you create one ball for each finger your screen can recognize; then you enable drag-and-drop with the built-in `Sprite` methods. You can drag each ball with a different finger simultaneously.

EXAMPLE 23-4 http://actionscriptbible.com/ch23/ex4

Multitouch Drag-and-Drop

```
package {
  import flash.display.Sprite;
  import flash.events.TouchEvent;
  import flash.ui.Multitouch;
  import flash.ui.MultitouchInputMode;
  public class ch23ex4 extends Sprite {
    public function ch23ex4() {
      try {
        var test:Class = Multitouch;
        if (Multitouch.supportsTouchEvents) {
          Multitouch.inputMode = MultitouchInputMode.TOUCH_POINT;
          for (var i:int = 0; i < Multitouch.maxTouchPoints; i++) {
            var b:Ball = new Ball();
            b.x = Math.random() * stage.stageWidth;
            b.y = Math.random() * stage.stageHeight;
            addChild(b);
          }
          stage.addEventListener(TouchEvent.TOUCH_BEGIN, onTouchBegin);
          stage.addEventListener(TouchEvent.TOUCH_END, onTouchEnd);
        } else {
          trace("Sorry, your device doesn't support touch-level events");
        }
      } catch (error:ReferenceError) {
        trace("Sorry, but multitouch is not supported in this runtime.");
      }
    }
    protected function onTouchBegin(event:TouchEvent):void {
      var b:Ball = event.target as Ball;
      if (!b) return;
      b.startTouchDrag(event.touchPointID, false);
    }
    protected function onTouchEnd(event:TouchEvent):void {
      var b:Ball = event.target as Ball;
      if (!b) return;
      b.stopTouchDrag(event.touchPointID);
    }
  }
```

```
}
import flash.display.BlendMode;
import flash.display.Sprite;
class Ball extends Sprite {
  public var touchPointID:int;
  public function Ball() {
    graphics.beginFill(Math.random() * 0xf0f0f0);
    graphics.drawCircle(0, 0, 70);
    blendMode = BlendMode.MULTIPLY
  }
}
```

Using the Accelerometer

Many mobile manufacturers are including accelerometers in their handsets. This sensor detects motion and orientation by sensing its acceleration relative to freefall; it's only really useful for handheld devices, not so much for desktops that stay put on the floor or a desk.

Accelerometers are already a big part of gaming today — they are used for many games on mobile phones, and they enable orientation-sensing controllers like the Wiimote and DualShock 3. They can be used for more passive interaction, where you don't have to be positioned to accurately touch the screen, or as a supplemental input method. Device motion can also be interpreted for another level of gestural input, like the "shake" gesture common on iPhone and iPad, in which a quick shake of the phone usually means "undo" or "reload." Other gestures are certainly possible.

With Flash Player 10.1, you can access data from supported accelerometer devices. Much like `Camera` or `Microphone`, covered in Chapter 33, "Capturing Sound and Video," you use static methods of `Accelerometer` (in the `flash.sensors` package) to configure the device; then you obtain an instance of `Accelerometer` to interact with.

Because the `Accelerometer` class was added in Flash Player 10.1, if you want to make sure the code works with older versions of Flash Player, you should wrap code that uses it in a `try/catch` block, as you did with `Multitouch` earlier in this chapter. If the `Accelerometer` class is present, find out if a compatible device is installed with the static accessor `Accelerometer.isSupported`.

```
try {
  var test:Class = Accelerometer;
  if (Accelerometer.isSupported) {
    var acc:Accelerometer = new Accelerometer();
    acc.addEventListener(AccelerometerEvent.UPDATE, onAccelerometerData);
  }
} catch (error:ReferenceError) {
  trace("Sorry, but accelerometers are not supported in this runtime.");
}
```

Now all you have to do is get a concrete `Accelerometer` instance — simply call its constructor — and subscribe to its `AccelerometerEvent.UPDATE` event.

When you attempt to use an `Accelerometer` instance, the device may prompt the user for permission, or use her saved privacy settings, before giving you access. You'll know you've been denied when `Accelerometer.isMuted` is `true`.

However, if all goes well, now you should be receiving `AccelerometerEvents`. If you wish, you can tweak the frequency of this data by using the static method `Accelerometer.setRequestedUpdateInterval()`. Each event object contains a vector representing measured acceleration, as three components: `accelerationX`, `accelerationY`, and `accelerationZ`. These are measured in Gs and are relative to the phone when you hold it in front of you with the screen facing you (and the y-axis is up, unlike the screen space where positive y is down). In this orientation, if you hold the phone still, `accelerationY` should be −1, because gravity is pulling down on the phone at the rate of 1G. If you drop your phone, it's in freefall, and the acceleration vector is <0, 0, 0> until it spikes up very high and you curse very loudly.

With these three values and some vector math, you can control orientations in 3D. You can isolate a single value and look for a pattern such as a shake. If you're going to do any direct mapping of accelerometer input, you will benefit from some data smoothing algorithms such as low-pass filtering.

Other Sensors

The `flash.sensors` package may become the home to more sensors in future versions of the Flash Player API. There is already one more that, although it's not currently accessible to Flash Player, can be used in AIR 2.0 and in native-compiled iPhone apps and may open up to Flash Player in the future. The `Geolocation` sensor class provides access to location services on a device, such as a GPS. It works in much the same way as `Accelerometer`: create a new `Geolocation` instance and subscribe to its `GeolocationEvent.UPDATE` event. The `GeolocationEvent` object contains all you need to know about the host device's location, between `latitude`, `longitude`, `altitude`, `heading`, and `speed`.

Summary

- Multitouch screens and sensors can enable an application to be more intelligent and react to the user in a more meaningful or natural way. Flash Player 10.1 and later have access to a wide array of devices: keyboard, mouse, touchscreen, camera, microphone, and accelerometer.

- For backward compatibility, you may want to test for existence of FP10.1-only classes, catching `ReferenceErrors` and falling back if they arise.

- Multitouch capabilities depend not only on hardware, but on the host OS and drivers. Research the capabilities of a multitouch interface with the `flash.ui.Multitouch` class.

- You can use multitouch in Flash Player in one of three mutually exclusive modes, set with `Multitouch.inputMode` to a constant of `MultitouchInputMode`.

- In touch mode, you get all finger tracking events with `TouchEvent`. It's more flexible, and you can define your own behavior.

- In gesture mode, the OS interprets sets of touches into common gestures. You get the benefit of a common vocabulary at the expense of customization. Gestures are broadcast as instances of `GestureEvent` and its subclasses.

- Accelerometer support gives you access to the orientation and motion of the device in three dimensions, when the appropriate sensor is present.

- Create an `Accelerometer` instance and subscribe to its `AccelerometerEvent.UPDATE` event to get accelerometer data, an event object with the vector encoded in `accelerationX`, `accelerationY`, and `accelerationZ` properties.

Part V

Error Handling

Errors and Exceptions

Despite our best intentions and wishes, programs sometimes fail. In Chapter 5, "Validating Your Program," you learned about ways the compiler can catch your typos and invalid requests: syntax errors and type mismatches. These are kinds of compile-time errors. But there are countless kinds of errors that can happen while your program is running, and these may be the result of a logical error in your code or just the result of unexpected circumstances out of your control. Whichever is the case, ActionScript 3.0 gives you the opportunity to handle these errors at runtime. This chapter provides the tools for you to understand runtime errors and keep them from affecting the end user.

FEATURED CLASSES

```
Error

ArgumentError

IOError

ReferenceError

SecurityError

TypeError

flash.error.*

flash.events.ErrorEvent

flash.events
  .UncaughtErrorEvent
```

Comparing Ways to Fail

If a certain function fails, there are a few ways for it to signal this failure. Some functions return a value that indicates that the function did not complete successfully. Code that uses this function must then check to make sure all went as planned before continuing. This approach has a few problems, though.

First, returning a Boolean signifying whether the function call was successful is not always possible. What if the function is already supposed to return a value? If you are returning a complex object, setting the value of this returned object to `null` is one option. If you are returning an `int`, you must decide on a reserved number to signify failure, such as `-1`. But these values might actually be valid responses, which also means that different functions might return different values for a failure state. Checking for failure becomes more and more complex this way.

The increasing complexity of checking return values of these functions is another good reason to avoid this approach. If you have to wrap every function call in an `if` statement to check its results, your code starts becoming ugly and unnecessarily complex.

Most programmers respond to this dilemma by skipping these checks, assuming that when things go wrong, they can insert the checks only where really necessary. In fact, it's sometimes not obvious that the function is supposed to return a particular value if it fails. It's not entirely the programmer's fault for being negligent here: it's difficult to keep track of every possible failure point.

The ultimate result of this kind of error reporting is a problem that might be familiar to you. Say you have a sequence of operations: you call a function, assign the returned value to some variable, use that variable in a different context, pass it around, and do some more things to it. Eventually, way down the line, something truly fatal happens. When you finally catch it, you have no way to know that the problem is from seven steps ago, when a function returned `null`, which was automatically converted to `false`, or an empty string, or the number `0`, and the program went on assuming this was the intended value. It becomes your job to painstakingly tease out what went wrong and where.

That's why this kind of error reporting is known as *failing silently*. Unless you put in those checks yourself, the program will keep on truckin', attempting to use your failure value as a real value, as long as it's nonfatal. And when everything fails silently, everything is nonfatal, so you really don't find out what went wrong until much later. As a general strategy, you should make your code noisy. Be alarmist! Run in circles, scream, and shout! In code, as in life, the sooner you become aware of a potential problem, the easier it is to fix it.

I should draw a distinction between ActionScript language features and application design. Returning `null` and using exceptions are two ways to signal and handle errors. These are part of ActionScript's error-handling capabilities. Knowing what to do when a runtime error occurs is part of an error-handling *strategy* and is part of your application design. In this chapter, I'll cover low-level error handling; once you have this as a foundation, you learn how to employ these practices in an error-handling strategy in Chapter 26, "Making Your Application Fault-Tolerant."

Understanding Exceptions

Exceptions are another way to signal, transmit information about, and capture errors at runtime. They are a language feature made explicitly for this purpose, so they won't interfere with your code's normal activities. They don't prevent you from returning certain values or limit anything your code can do.

Exceptions are both ordinary and extraordinary. They are objects that exist in the virtual machine and can be manipulated like any other object. But when you use exceptions, you also alter the control flow of your program.

Because exceptions are objects, you can consider them nouns. To understand them better, let's examine what verbs you can apply to them. You might say that exceptions are like baseballs when you're the fielding team. You can throw them and catch them, but if nobody catches them, it's a real error (ha). You can *throw* an exception, and you can *catch* an exception. There are also *uncaught* exceptions, which are unarguably a bad thing.

Throwing Exceptions

Errors at runtime are represented by exceptions; throwing an exception is the way in which you signal an error. Confusingly, there is no `Exception` class. Rather, all exceptions are represented by instances of `Error` or its subclasses. Further, there is special syntax for throwing an exception. You are performing a verb and have a noun, so you might expect this to look like the following:

```
var myException:Error = new Error();
myException.throw();
```

But we have the actor wrong here. Really, the exception represents the error, and the virtual machine handles propagating it; the exception is passive, much like an event object. The syntax for throwing an exception is this:

```
throw new Error("oh noes!! i already eated the cookie", 512);
```

You can use two lines, as in the first snippet, to create and then throw the exception, but using one line as in the preceding code is more common. As shown in the preceding code, `Error` objects are allowed to carry a human-friendly message describing what went wrong in more detail, and a more computer-friendly error code that you can use in your error-handling strategy. In addition, the type of the exception encodes information about what went wrong. There are several subclasses of `Error`, and you can define your own. Both of the exception's descriptors, the `message` and the `id`, are optional. In this snippet, an exception is thrown without a description, but its type communicates some information:

```
throw new ArgumentError();
```

Note
The implementation of exceptions in ActionScript 3.0 is similar to that in Java, but methods that throw errors do not explicitly say so in the method signature as they do in Java. ∎

Catching Exceptions with Try and Catch

When you're handling errors with exceptions, you can handle errors and do whatever is necessary to recover from them by catching the appropriate exception. There's more to catching exceptions than a single-line catch method, however. Handling exceptions involves surrounding the dangerous code in a special kind of construct. This construct has at least two parts: first, the code that might generate exceptions, and next, one or more blocks to handle certain kinds of exceptions; and optionally a block to run whether there were exceptions or not.

These three units make up a `try/catch/finally` block, or simply a `try/catch` block because the `finally` clause is optional. One of the benefits of exceptions is the way in which they can break out of the normal flow of a program. When an expression results in a thrown exception, Flash Player drops everything and starts looking for an appropriate `catch` block to handle the exception. It doesn't continue to execute the next line of code. This is in stark contrast to handling errors by returning `null`, in which case the code keeps executing, making it more likely to lose track of where the error occurred. This also enables you to write code that otherwise assumes that everything went well, confident that the lines after an error will not be executed.

You encapsulate potentially exception-generating code and the instructions that depend on it in its own `try` block. Everything that does not depend on the potentially error-prone operation and has no chance of failure can go outside the block. Everything that might cause an error, and the code that depends on an error not happening, goes inside the block. Here's an example of a `try` block, but note that it is not complete (it won't even compile) without the `catch` block, which I cover next:

```
var foo:Array = [1, 2, 3];
try
{
    var a:int = potentiallyUnsafeOperation(foo[0]);
    a += 10;
    trace(a);
}
//TODO: catch goes here
var b:int = 1 << 10;
```

Before and after the `try` block, you do operations completely unrelated to the potentially unsafe function, which might throw an exception. Inside the `try` block, however, you include everything that depends on that function executing properly: anything that uses its return value `a`. If `potentiallyUnsafeOperation()` threw an exception, or if `foo` happened to have no object in index 0, or if that object were of the wrong type for the method's parameter, it would never store the return value into `a` because the exception would have interrupted execution.

You use `try` blocks not just to contain an error, but to define where it should be handled. Each `try` block must be coupled with one or more `catch` blocks. If the exception is generated in a `try` block and handled in a subsequent `catch` block, execution can resume after the block is over. Thus, the exception is generated and handled, and then the program keeps going. *Voilà!* Furthermore, with multiple `catch` blocks, you can handle multiple types of errors differently within a single `try`/`catch` block, allowing your `try` block to contain a whole series of actions, each action depending on the success of the one before it, even if they all throw different kinds of errors.

Let's add a `catch` block to the preceding snippet:

```
var foo:Array = [1, 2, 3];
try
{
  var a:int = potentiallyUnsafeOperation(foo[0]);
  a += 10;
  trace(a);
} catch (error:Error) {
  trace("An error occurred", error.message);
}
var b:int = 1 << 10;
```

If the potentially unsafe function call ends up throwing an exception, flow proceeds immediately to the `catch` block. There, the system traces out an error message, including the message defined in the `Error` object. Finally, execution continues with the first line outside the block, assigning 1024 to `b`.

The Exception Flow

When you throw an exception, it follows a well-defined flow, much like events do. Exceptions bubble up through the call stack until they find themselves in a `try` block. For each one of these, the exception compares its type to the types of the errors the associated `catch` blocks are looking for, top to bottom. If no `catch` block catches that type of exception, it continues its way up the call stack. If it reaches the top of the call stack without being caught, it becomes an uncaught exception.

This means that not only do you have a choice of where to catch exceptions, but that choice makes a difference in how your program behaves. Take, for example, the class shown in Example 24-1, which computes 10 square roots of whole numbers between −50 and 50.

`http://actionscriptbible.com/ch24/ex1`

Throwing and Catching an Exception

```
package {
  import com.actionscriptbible.Example;
  public class ch24ex1 extends Example {
    public function ch24ex1() {
      try {
        for (var i:int = 0; i < 10; i++) {
          var n:Number = Math.round(Math.random() * 100 - 50);
          trace("√"+n+" = "+squareRoot(n));
        }
      } catch (err:ArgumentError) {
        trace("ERROR: " + err.message);
      }
    }
    protected function squareRoot(n:Number):Number {
      if (n < 0) {
        throw new ArgumentError("squareRoot() doesn't support \
imaginary numbers.");
      }
      return Math.sqrt(n);
    }
  }
}
```

The `squareRoot()` method checks its input and considers an attempt to calculate the square root of a negative number an error. Checking your inputs for validity is one important part of an overall error-handling strategy. The virtual machine and compiler can ensure your arguments are of the right type, but sometimes your preconditions must be more specific. In this case, the best the type system can do is ensure that a `Number` is passed to `squareRoot()`, but it can't verify that the number is positive with types alone. An `ArgumentError` is an `Error` subclass designated for exceptions relating to arguments passed to a method.

In any case, this version of the example will terminate the loop the first time a negative number is chosen. You might receive only three square roots before an error message, or seven, or none, depending on your luck. This says that the entire loop is predicated on all the iterations finishing successfully.

However, you can change this behavior by simply moving the `try` block. By putting the `try` block inside the loop body, the program always prints out 10 results or errors rather than quitting after the first error. Why don't you modify the code and *try* it? (Groan.) The flow of control in your program is affected by exception handling, and changes in your exceptions can change the outcome of your program.

The `catch` blocks specify what kind of error is being handled. Because all exceptions are either an instance of `Error` or a subclass thereof, catching an `Error` type handles *any* kind of exception. You can use this in the same way you would use `default` in a `switch` statement. Example 24-2 shows an error being handled further up the call stack, and with multitiered `catch` blocks.

Errors and the Call Stack

```
package {
  import com.actionscriptbible.Example;
  public class ch24ex2 extends Example {
    public function ch24ex2() {
      try {
        var n:int = sumSomeNumbers();
        trace(n);
      } catch (e:UnrelatedError) {
        trace("something unrelated happened.");
      } catch (e:DispleasingNumberError) {
        trace("somewhere, a number is displeasing.");
      } catch (e:Error) {
        trace("something bad happened");
      }
    }

    protected function sumSomeNumbers():int {
      var sum:int = 0;
      for (var i:int = 0; i < 10; i++) {
        sum += someNumber();
      }
      return sum;
    }

    protected function someNumber():int {
      var n:int = Math.round(Math.random() * 10);
      if (n < 2) {
        throw new PatheticallySmallNumberError();
      }
      return n;
    }
  }
}
class DispleasingNumberError extends Error {}
class PatheticallySmallNumberError extends DispleasingNumberError {}
class UnrelatedError extends Error {}
```

With any luck, this example will eventually throw a PatheticallySmallNumberError because it is displeased with encountering the number 0 or 1. This exception originates in someNumber(), goes unhandled to the code that called it in sumSomeNumbers(), which again is not in a try block, and finally goes up to where the call originated, when the constructor CatchBlocksExample() called sumSomeNumbers(). This is inside a try block, so the exception compares itself with the handled exception types in the associated catch blocks in order. Because the runtime is comparing the exception object to different classes to see if it is an instance of the class specified in the catch clause or its subclasses, you can think of this comparison as using the is operator. The PatheticallySmallNumberError is not an UnrelatedError, but it is a

`DispleasingNumberError`, so the second `catch` block will handle it, and "somewhere, a number is displeasing" is traced out. If neither of these `catch` blocks had caught the exception, again, the third would have, because all exceptions are `Errors`.

Uncaught Exceptions

When an exception is thrown without being caught anywhere, it is an *uncaught exception*. All your chances to intercept and handle it, by definition, have passed, and it is up to Flash Player to decide what to do with the error. Different versions of Flash Player treat uncaught exceptions differently.

If you're developing applications, you're probably running SWF files in the debug version of Flash Player. When an uncaught exception occurs in the debug version of Flash Player, execution halts and you are presented with a dialog box giving you information about the error, as shown in Figure 24-1.

FIGURE 24-1

An uncaught exception using the Debug Flash Player

Even better, if you are running a SWF compiled for debugging, and you have connected to a debugger using Flash Builder, or Flash Professional, instead of the dialog box, your IDE will enter an interactive debugging mode. Debugging applications this way is covered in greater detail in Chapter 25, "Using the AVM2 Debugger and Profiler." Suffice to say, this gives you an excellent way to identify bugs immediately, and at their source.

If you are running the production or release distribution of Flash Player, unhandled exceptions are silent. No dialog box is displayed, the currently executing code is immediately terminated, and the next frame is rendered, at which point further code might be triggered. This means that if you should somehow let a bug slip into your program unhandled, and that program is deployed, the end users will not see a glaring "Error" dialog box as you do. The program might not work correctly, but that's not nearly as worrisome as a giant error dialog box. When this happens to you — and it will — explain this to your boss, customer, or client, say that everything is going to be all right, and go fix that bug.

The finally Clause

Finally, we come to `finally`. The optional `finally` clause can appear after the `catch` blocks, and code inside it will execute after the `try` block successfully completes or after the relevant `catch` block finishes. If the `try` block depends on certain objects that need to be set up and created, use

the `finally` block to break them down and clean them up. They should be properly disposed of whether you had an error or not.

Most of the examples that necessitate a `finally` block involve creating a file handle or stream object, trying to read from it, and then closing it whether there was an error or not, because these kinds of objects need you to explicitly clean up after them. However, most of these accesses are asynchronous in ActionScript 3.0, and `try/catch/finally` helps you catch synchronous errors only, as Example 24-3 shows.

EXAMPLE 24-3 `http://actionscriptbible.com/ch24/ex3`

Cleaning Up After an Error with finally

```
package {
  import com.actionscriptbible.Example;
  import flash.errors.IOError;
  import flash.net.Socket;

  public class ch24ex3 extends Example {
    public function ch24ex3() {
      try {
          var s:Socket = new Socket();
          s.connect("http://www.actionscriptbible.com/", 80);
          //TODO: add asynchronous error handling;
          //see "Handling Asynchronous Errors" below.
          trace("SUCCESS!");
      } catch(error:IOError) {
          trace("I/O Error:", error.message);
      } catch(error:SecurityError) {
          trace("Security Error:", error.message);
      } finally {
          s.close();
      }
    }
  }
}
```

This is a good example of an object that needs to be cleaned up after, and thus a good use of `finally`. However, even this is not entirely realistic because you'd usually do many (asynchronous) accesses on the socket before closing it, so you wouldn't have the opportunity to wrap the entire session in a single block.

Rethrowing Exceptions

If the baseball analogy holds true, you can also pass the ball: catch it and throw it to another team-mate. If you catch an exception and determine, by examining the `id` of the `Error`, perhaps, that a higher authority needs to handle the exception, you can rethrow the same `Error` object that was

passed to you in the first place. This puts the exception back in the flow up the call stack, as you can see in the following snippet:

```
try
{
    try
    {
        throw new Error();
    } catch (error:Error) {
        trace("inner handler");
        throw error;
    }
} catch (error:Error) {
    trace("outer handler");
}
//inner handler
//outer handler
```

Errors Generated by Flash Player

In ActionScript 3.0, many of the built-in methods throw exceptions to signal errors. This is a godsend because it helps you track down bugs immediately as they surface. If you don't handle them you are alerted via the uncaught exception handler (the error dialog box or debugger).

You can catch these errors in the same way you catch any other exception. Wrap the potentially error-causing code and any dependent code in a `try` block, and create `catch` blocks for the kinds of errors that may arise.

A common time to generate errors is when downcasting objects. If you try to convert a type to an incompatible type using an explicit cast, a `TypeError` exception is thrown:

```
var displayObject:DisplayObject;
var object:Object = "Definitely Not a Display Object";
try
{
    displayObject = DisplayObject(object);
} catch (error:TypeError) {
    trace("Incompatible cast!");
    displayObject = new Sprite();
}
addChild(displayObject);
```

In this case, an incompatible cast would recover by creating a new, empty `Sprite` to add to the display list. This way, the code can continue executing.

To find out what kinds of exceptions a given method can throw, look up its definition in the ActionScript 3.0 Language Reference, and look for the "Throws" header. The documentation includes all exceptions that each method can throw, and descriptions of the errors. An exhaustive list would be difficult to reproduce here and of little use. However, Table 24-1 lists quite a few common exceptions.

TABLE 24-1

Built-In Exception Types

Exception Type	Description	Potential Cause and Notes
`Error`	Base exception type.	Custom errors without custom subclasses. Not abstract; can be used for actual errors.
`EvalError`	Errors in `eval()`.	Any call to `eval()`, which is provided for ECMAScript compatibility but not implemented.
`RangeError`	A number is out of the valid range.	Converting numeric types, getting display object children at invalid indices, creating `Array`s of invalid sizes, writing bytes as characters when they are not in the range of acceptable characters, and so on.
`ReferenceError`	Attempt to access an undefined property of an object.	Looking up a property that doesn't exist on a sealed class.
`SyntaxError`	Code is not syntactically valid.	Usually compiler catches syntax errors, but they can arise with invalid `RegExp`s.
`TypeError`	Mismatched types.	The type of an argument is incompatible with the expected type, as a parameter to a function, operator, assignment, and so on.
`ArgumentError`	Error with arguments passed to a function.	Common in methods with variable argument lists or flexibly typed arguments such as `Object` or `*`. Occurs when function does its own validation manually.
`SecurityError`	Violation of Flash Player's security policy.	When access is denied to a URL, server, or system hardware, either by the security sandbox or the user.
`VerifyError`	Malformed SWF encountered.	Attempting to load a corrupt SWF into the program.
`flash.error.EOFError`	Attempted to read beyond the end of the stream/file.	When reading more bytes than are available from a `URLStream`, `ByteArray`, or `Socket`.

Exception Type	Description	Potential Cause and Notes
`flash.error .IllegalOperationError`	A method call exists but is not supported.	Calling certain `DisplayObjectContainer` methods on stage, `FileReference` methods called in the wrong order, `Accessibility` methods are called when the SWF is not compiled with accessibility support, and so on.
`flash.error.IOError`	Error reading or writing to a resource.	Can happen to any network access, because networks are inherently unreliable.
`flash.error .MemoryError`	Memory was requested that cannot be provided.	Requesting more memory than can be addressed. More common on embedded devices where addresses might be shorter. When more memory is requested than is available, a different error (#1000) is thrown.
`flash.error .ScriptTimeoutError`	Code executes beyond a set duration before completing and allowing Flash Player to draw the next frame.	The script timeout duration can be written into the SWF. By default, it is 15 seconds. Can be caught only once; otherwise, it is an uncatchable error.
`flash.error .StackOverflowError`	The execution stack exceeds its maximum depth.	A recursive function is probably not reaching the intended end conditions and recurses infinitely.

Custom Exceptions

Because `catch` blocks depend on the type of the exception object to match and handle exceptions, you should use the appropriate kind of `Error` object for each error. For some errors that you might encounter in your own code, the built-in `Error` classes are appropriate, such as throwing `ArgumentErrors` and `RangeErrors` when arguments are invalid or out of range.

You should use custom `Error` subclasses when you are throwing errors specific to your program that you are going to handle. As you have already seen in this chapter, this is as simple as creating an otherwise empty subclass of the `Error` class. Additionally, you can create more specific

exceptions by further extending `Error` subclasses. This hierarchy can be used to `catch` with more specificity.

For example, you could enable *assertions* in your code easily. An assertion is a basic defensive-coding tool that simply requires some expression to be true. You use it to make sure that pre-conditions and postconditions of your methods are upheld, and any time you need a sanity check.

```
function assert(b:Boolean):void { if (!b) throw new AssertionError(); }
class AssertionError extends Error {}
//usage
assert(inputNumber >= 0);
Math.sqrt(inputNumber);
```

If you want to share these outside the file they're defined in, you can put the `assert()` method and the `AssertionError` class in their own files inside some package, and make both items public.

Any time the assertion fails, an exception is thrown. Appropriately, this is an `AssertionError`, so that catch clauses can specify it by class name. (Although I can't think of a case where it's a good idea to catch an assertion error. Assertions are there to notify you, the programmer, and they do no good if they're handled.)

Handling Asynchronous Errors

Not all errors occur immediately, just as not all functions can return a result immediately. ActionScript 3.0 has many functions that perform their task asynchronously, reporting their result by firing an event. Unfortunately, because exceptions are tied into the control flow and asynchronous calls may finish after the code inside a `try` block is done executing, exceptions are not an appropriate mechanism to handle asynchronous operations that result in failure.

Instead, asynchronous operations signal failure using the same method by which they return a valid result: by broadcasting an event. These error events are broadcast and subscribed to in the same way as any other event. For a review of events, please reference Chapter 20, "Events and the Event Flow."

Because asynchronous errors are just events, they are not subject to the exception flow. You must subscribe to the error events of asynchronous methods if you wish to handle them. An uncaught asynchronous error event gets the same treatment in the debug Flash Player as an uncaught exception: an alarming dialog box. However, because these errors are asynchronous, the debug player won't break into the debugger to show you where your error is; the error is where you forgot to add a handler for the error event.

To see what asynchronous errors are dispatched by built-in methods, check the AS3LR for that method and look under the "Events" header. Asynchronous error events typically have names ending in `ErrorEvent`, such as `IOErrorEvent`. In Example 24-4, you add an error event listener to the socket from Example 24-3.

`http://actionscriptbible.com/ch24/ex4`

Asynchronous Error Events

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  import flash.events.IOErrorEvent;
  import flash.events.SecurityErrorEvent;
  import flash.net.Socket;

  public class ch24ex4 extends Example {
    protected var sock:Socket;
    public function ch24ex4() {
      sock = new Socket();
      sock.addEventListener(Event.CONNECT, onSocketOpen);
      sock.addEventListener(IOErrorEvent.IO_ERROR, onIOError);
      sock.addEventListener(SecurityErrorEvent.SECURITY_ERROR, onSecurityError);
      sock.connect("www.actionscriptbible.com", 80);
    }
    protected function onSocketOpen(event:Event):void {
      trace("Connected succesfully!");
      sock.close();
    }
    protected function onIOError(event:IOErrorEvent):void {
      trace("ERROR:", event.text);
      sock.close();
    }
    protected function onSecurityError(event:SecurityErrorEvent):void {
      trace("ERROR:", event.text);
      sock.close();
    }
  }
}
```

This example attempts to connect to a web server, an asynchronous operation. Failure is signaled by broadcasting the `IOErrorEvent.IO_ERROR` event, which is captured and handled by `onIOError()`. Likewise, a security violation broadcasts the `SecurityErrorEvent` `.SECURITY_ERROR` event and triggers `onSecurityError()`.

Capturing Unhandled Events

When an exception is not handled by any `catch` block in the call stack, or an error event isn't handled by any event listener, Flash Player aborts executing ActionScript until the next frame, and in the debug player, pops an error dialog box. In Flash Player 10.1, you can register a global error handler that catches these wayward errors that fall through every other net. This is your last line of defense against the unhandled error dialog box.

Version

FP10.1. Global event handlers are only available in Flash Player 10.1. ■

With great power comes great responsibility, and the global error handler is not simply there to absolve you of good error handling. The fact is, an unhandled exception has already done its damage. There's not much you can do if your program has already failed to catch the error.

What a global error handler does help with is logging and reporting the error, and preventing that damnable dialog box. You'll learn more about what to do with unfixable errors in Chapter 26.

To register a global event handler, listen to the `UncaughtErrorEvent.UNCAUGHT_ERROR` event dispatched by the `UncaughtErrorEvents` object owned by the SWF's `LoaderInfo` object. You'll learn more about `LoaderInfo` in Chapter 27, "Networking Basics and Flash Player Security," but it's not necessary to understand now. All you need to do is add the global error handler like so:

```
loaderInfo.uncaughtErrorEvents.addEventListener(
    UncaughtErrorEvent.UNCAUGHT_ERROR, onUncaughtError);
```

where `onUncaughtError()` is your global error handler function. Write this code inside any `DisplayObject` attached to the display list of the SWF generating errors.

Summary

- Returning a special value to represent errors is an approach with severe limitations.
- Exceptions are a better way to report and handle errors.
- Exception objects represent an error.
- Exceptions can be thrown, caught, rethrown, or uncaught.
- Exceptions interrupt the flow of control of your program.
- Exceptions travel up through the call stack until they find themselves in a `try` block, and they match the first `catch` block that handles exceptions of the same type.
- `Error` is the base class for exceptions and the catch-all type for `catch` blocks.
- The type of exceptions denotes the kind of error, and its `id` and `message` transmit more information.
- Uncaught exceptions are caught in the debugger, display an error dialog box in the debug Flash Player, or stop all frame scripts in the release Flash Player.
- You can create your own `Error` subtypes to use as custom exceptions.
- Flash Player defines many built-in types of exceptions.
- Asynchronous errors are just events.

Using the AVM2 Debugger

Every builder needs the right tools for the job, and with ActionScript 3.0 comes a much-improved tool for correcting problems: an interactive debugger. With effective use of the debugger, you can locate and correct problems in your code intelligently, without hunting or head-scratching.

Introducing Debugging

An interactive debugger performs lots of different jobs. The net effect is that you can run your program in a controlled environment and interactively follow its execution. Without a debugger, executing code is like running an experiment with tiny particles: because you can't see them, you have to rely on secondary or tertiary effects of your experiment to determine what's really going on. You must carefully craft situations in which you can measure the outcome and attempt to support your theory with the data. When you have a debugger, it's like having a powerful microscope: you can see everything in perfect detail, and make primary observations rather than deducing cause-and-effect; it removes the mystery from your code.

When you compile your ActionScript 3.0 program, it turns into a series of simple instructions called bytecodes that are interpreted by the ActionScript Virtual Machine 2 (AVM2). When you run the program in Flash Player, it's those bytecodes that are being interpreted; the system doesn't know or care what you originally wrote in AS3 code. But when you compile your SWF with debugging information and use the debugger, you can watch the program being run by the AVM2 as if the virtual machine were running your original code and not the bytecode. You wrote the code, and the bugs are in your code, so you get to see your code. The debugger helps you visualize your code in action.

Running your program in the debugger is like stepping in as the director and running a dress rehearsal of your program. You have the script — the source code — in your hand. But now you can call the shots. You can start and stop the production. You can stop at a specific scene, or a specific line, and have your

actors read the scene line by line. You can automatically cut the production when a catastrophe occurs. And you have a camera that lets you focus in on any actor. You can always find where the production is in the script, and you can always find out how it got there. Using your powers as director, you can interactively run through your production and work out the kinks in the script.

You'll see how this metaphor applies as you learn about the features of the interactive debuggers.

Launching the Debugger

The first thing you need to do to debug your program is to get the debugger running. Different products that use ActionScript 3.0 have different options for debugging, which I cover in this section. At a minimum, to start a debugging session, you need three things:

- A SWF compiled specifically for debugging
- The debugger version of Flash Player
- An AVM2 debugger

These loose requirements mean that you can debug programs in a variety of situations. You can debug a program running inside your browser from a remote web site, for instance, as long as your browser is running the debugger version of Flash Player, the remote SWF has been compiled for debugging, and you are running a debugger locally.

Tip
It's possible to have different versions of the Flash Player installed in different browsers, as well as to have a different version of the standalone SWF player. If you are having trouble starting a debugging session, double-check the installed version of Flash Player in the environment you are attempting to debug the SWF in. Visit `http://actionscriptbible.com/version.swf` **to get information about your copy of Flash Player.** ∎

When you're developing an application on your computer and you run into a problem, you can usually run a debugging session locally, and if you're using Flash CS3 Professional and up, Flex Builder 2 and up, or Flash Builder, you can do this with one click or keystroke. So although there is a multitude of ways you can set up a debugging session, for most purposes it's a simple affair.

Each development environment for ActionScript 3.0 has its own debugger. If you are building your applications in Flash Professional, you will use the AVM2 debugger in Flash. If you are using Flex Builder or Flash Builder, you will use the Flex Debugging perspective or Flash Debug perspective. If you are using the Flex SDK toolchain, you can use the command-line Flash Player Debugger, fdb. To get started, let's look at how to launch a debugging session in the more recent tools: Flash CS5 Professional and Flash Builder 4 Premium.

Starting and Stopping the Flash Professional Debugger

You can start debugging a project in Flash Professional by choosing Debug Movie from the Debug menu, as shown in Figure 25-1, or by pressing Cmd+Shift+Return (on a Mac) or Ctrl+Shift+Enter (on a PC).

Doing this compiles your application with debugging information, launches the SWF in the debugger version of the Standalone Flash Player, and opens panels associated with debugging. You should be able to see and use the following panels: Debug Console, Variables, and Output. If any of these fail to appear, you can re-enable them by finding them in the Window ⇨ Debug Panels menu. It helps to organize the panels so that you can see all of them, as shown in Figure 25-2.

FIGURE 25-1

Starting a debugging session in Flash CS4 Professional
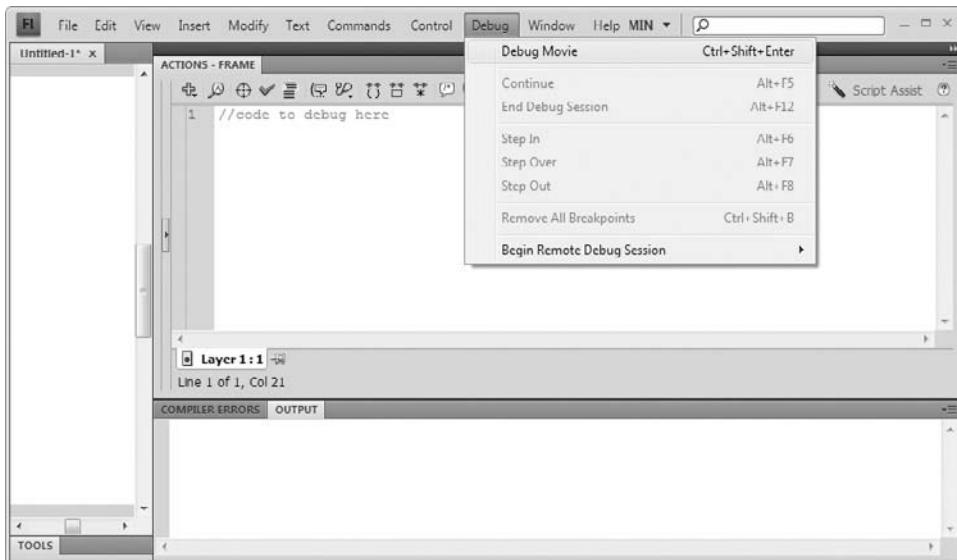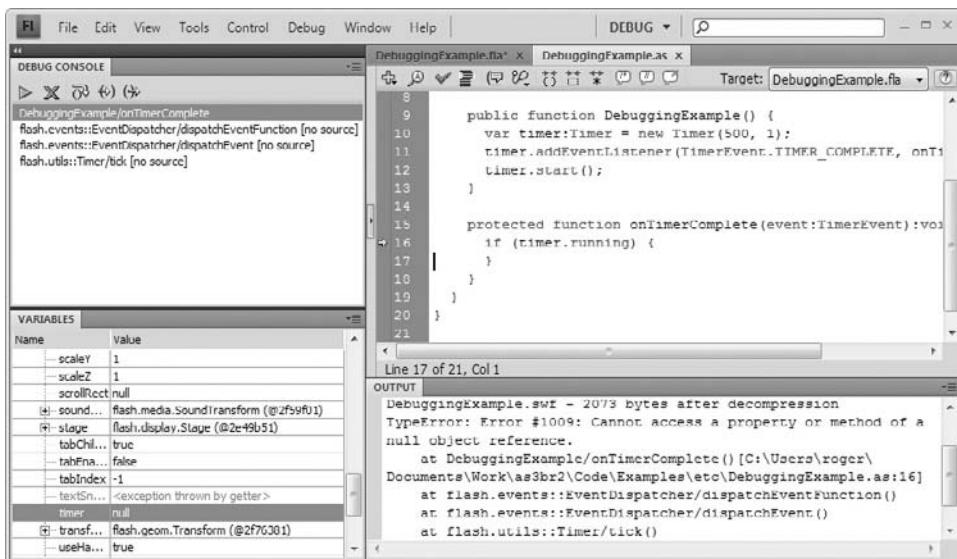
FIGURE 25-2

Debugging in Flash CS5 Professional

To stop a debugging session in Flash, select End Debug Session, either as the red X button in the Debug Console or under the Debug menu.
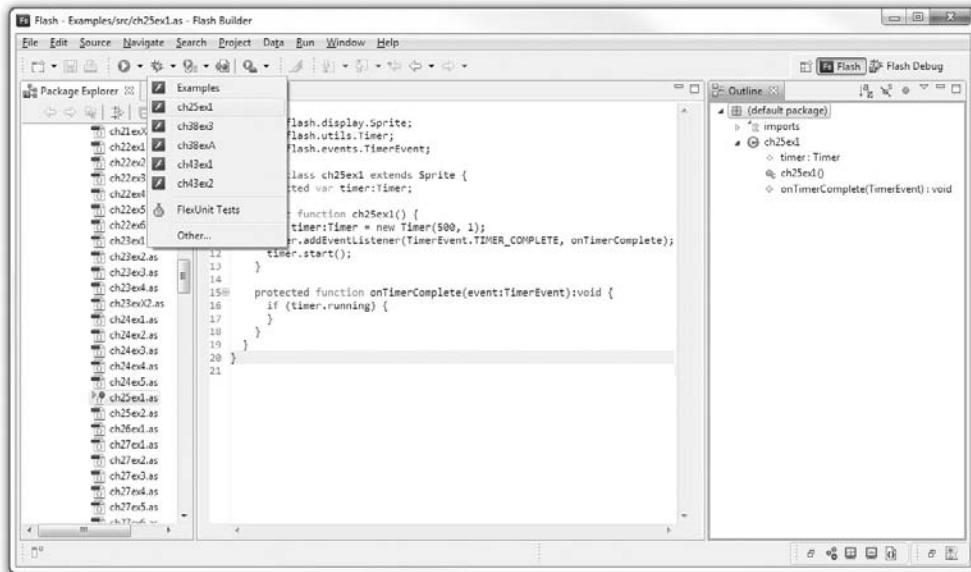
Starting and Stopping the Flash Builder Debugger

You can debug a project in Flash Builder by choosing Debug [*name of the application*] in the Run menu, by clicking the Debug button (a green bug icon in the toolbar), by setting up a custom debug configuration (choose Debug ➪ Other), or by pressing an associated keyboard command. Depending on your project's configuration, your project will launch in either a browser or a standalone Flash Player. If you want to debug in-browser, it's important to make sure that your browser has a debugging version of the Flash Player plug-in installed.

Once the debugging session starts, you may automatically be taken into the Flash Debugging perspective. If not, click the Flash Debug button in the perspective bar (by default in the upper-right toolbar). In Figure 25-3, a debugging session is about to be initiated in Flash Builder 4.
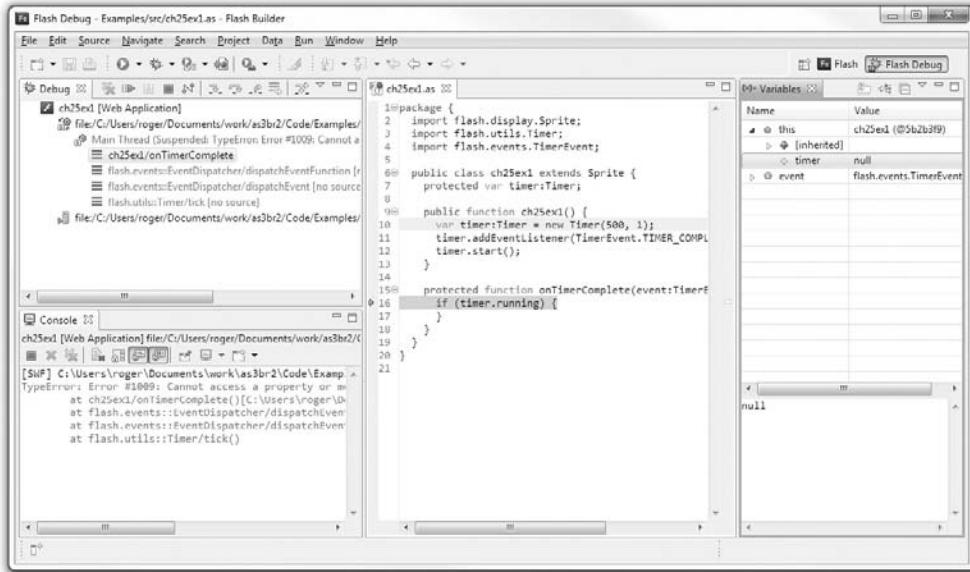
Starting the debugger in Flash Builder 4

In the Flash Debugging perspective, you should be presented with a few views: Debug, Variables, Console, Breakpoints, and Expressions. The Debug view is analogous to Flash's Debug Console, the Variables view to Flash's Variables panel, and the Console view to Flash's Output panel. If these views are not visible, you can enable them in the Window menu. Figure 25-4 shows the Flash Debugging perspective in action. The Expressions view is not pictured here.

FIGURE 25-4

The Flash Debug perspective in Flash Builder 4

To stop a debugging session in Flash Builder 4, select Run ➪ Terminate or press the Terminate button (the red square icon on the Debug view's toolbar). You can also disconnect the debugger from the Flash Player without terminating it by pressing the Disconnect button in the Debug view's toolbar.

Debuggers Compared

The Flash Builder and Flex Builder debuggers are, while essentially the same debugger as the Flash Professional debugger, easier to use and have a few additional useful features such as temporarily toggling all breakpoints (covered in the section "Taking Control of Execution"). If you have the choice, you might find it easier to debug your programs using the Flash Builder debugger.

Throughout the rest of this chapter, I use the Flash Professional debugger to illustrate debugging concepts. All the interactive debuggers, even the command-line debugger fdb, are operated in the same manner and support the same kind of actions.

Caution

While the debugger is running, either Flash Player or the browser remains in suspended animation, so it may appear to be an unresponsive program to the system. Instead of forcibly terminating the player or your browser, terminate the debugging session first. ■

Note

If you are debugging in a browser and you need to use the browser normally while debugging, use a browser that can run windows in separate processes, or debug in a different browser than you use for browsing. ■

Taking Control of Execution

To start a debugging session, you compile a SWF, launch it in a debugger version of the Flash Player, and connect a debugger to it. Although you're all set to debug now, nothing might happen. By default, the program runs normally. As the director, you need an opportunity to step in and yell "Cut!" before you can start working on a particular bug. This is called *breaking the program* (break as in "taking a break" more than "Honey, I broke that dachshund gravy boat you love"), or *halting execution*. There are three ways to transfer control between normal execution and interactive debugging.

Stopping at an Uncaught Exception

If your debugging session started properly, when things go catastrophically wrong, you will be handed the director's bullhorn and have an opportunity to step in. Uncaught exceptions, which without a debugger connected might pop up a dialog box, now automatically break the execution of the program right where the error occurs. Figure 25-2 shows a program halted because of an uncaught exception.

Halting everything when an uncaught exception occurs allows you to get to the cause of the error immediately, so with little effort, you can fix runtime errors as quickly as they crop up. The combination of an API that utilizes exceptions fully and a debugging environment that halts automatically on uncaught exceptions is great for finding and solving small bugs before they become big bugs. You saw how thoroughly the Flash Player API uses exceptions in the last chapter, "Errors and Exceptions."

In the remainder of this chapter, I look at how to use the tools the debugger gives you. For now let's take a sneak peek and see how you can use the debugger to resolve an uncaught exception. In Figure 25-2, you see the debugger immediately after an uncaught exception halted execution. The source code of this program is shown in Example 25-1.

EXAMPLE 25-1 `http://actionscriptbible.com/ch25/ex1`

An Uncaught Exception Halts Execution

```
package {
  import flash.display.Sprite;
  import flash.utils.Timer;
  import flash.events.TimerEvent;

  public class ch25ex1 extends Sprite {
    protected var timer:Timer;

    public function ch25ex1() {
      var timer:Timer = new Timer(500, 1);
      timer.addEventListener(TimerEvent.TIMER_COMPLETE, onTimerComplete);
      timer.start();
    }
```

```
    protected function onTimerComplete(event:TimerEvent):void {
      if (timer.running) {
      }
    }
  }
}
```

The arrow in the source editor in Figure 25-2 points to where the program was halted; this is where the exception must have originated. This line is displayed in bold in the preceding listing. The Output panel gives you the kind of error that occurred: `Cannot access a property or method of a null object reference`. You can put these two bits of information together: you tried to access a property or method of `null`, and it happened on a line where you check `timer.running`. At this point you might suspect that the timer reference is `null`, and you would be correct. In "Pulling Back the Curtain," you'll see how to use the Variables panel to investigate problems like this.

This runtime error, raised when you try to access something under a null reference, tells you what ultimately went wrong. You're likely to see this error when you erroneously assume that an object existed in the variable you're accessing. In some cases, `null` is a valid value for a given variable, in which case the error is simply assuming that the variable was non-`null`. In this case, `timer` shouldn't be `null`, so the cause for the error — looking for a property of a `null` variable — is a clue for finding the original logical error. Here, you fully expect `timer` to exist — after all, it's created in the constructor. So why is it `null`? When you set `timer` in the constructor, it was to a local `timer` variable, which shadowed the `timer` instance variable. By simply removing the local variable in the constructor, you can fix the error:

```
    //var timer:Timer = new Timer(500, 1); //original line with error
    timer = new Timer(500, 1); //corrected line
```

Let's introduce some techniques for using the debugger to answer questions like, "Why is it `null`?" With these tools, you'll be able to sleuth out the original cause of an error when the actual runtime error is only the outcome of a larger error.

Stopping at a Breakpoint

An essential technique for interactive debugging is setting breakpoints. A *breakpoint*, literally, is a point in the code at which normal execution will break and you can debug. In the stage rehearsal analogy, setting a breakpoint is like telling the actors that when they get to a certain point in the script, you're going to start directing them. You don't necessarily have to choose a line where you know an error occurs, either. You can set the breakpoint a few lines earlier so you can get some context, or you can set a breakpoint just to make sure a piece of code is behaving as you expect it to. Breakpoints are gateways between normal execution and debugging.

To set a breakpoint in an ActionScript file in Flash Professional, just click once on the left gutter (the gray vertical bar to the left of the line numbers) of the line you want to stop at. The breakpoint appears as a red octagon, like a stop sign. To toggle it off, just click it again. You can also use Debug ⇨ Toggle Breakpoint or its associated keystroke. To clear all breakpoints at once, Flash Professional provides Debug ⇨ Remove Breakpoints in This File and Debug ⇨ Remove Breakpoints in All AS Files.

To set a breakpoint in Flash Builder, double-click the gutter of the line you want to break at. The breakpoint appears as a blue circle. To toggle it off, double-click the blue circle, and it disappears. You can also use the menu item Run ➪ Toggle Line Breakpoint or its associated key command. You can manage breakpoints in Flash Builder with the Breakpoints view. This allows you to see all the breakpoints in one spot, selectively remove or temporarily disable them, and clear all of them at once.
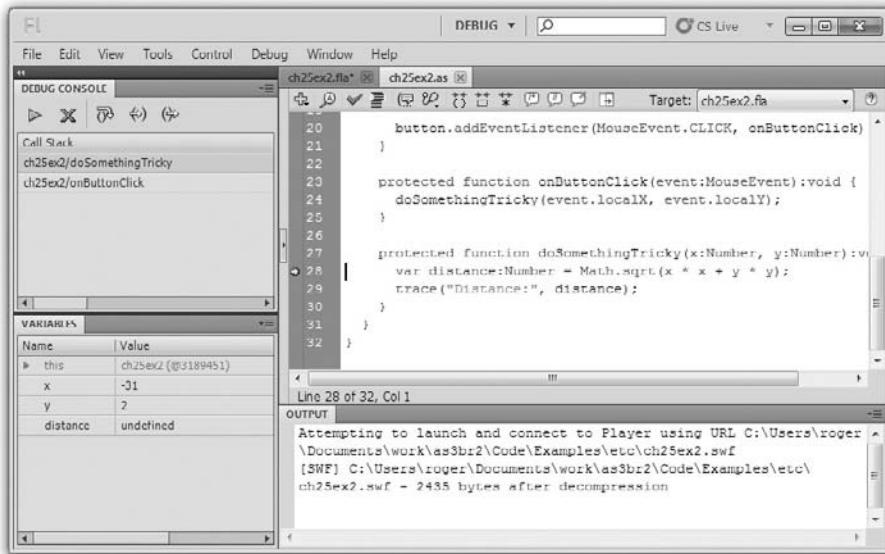
Tip

You may not set a breakpoint at a line where there is no executable code. You might also decide to break up a long expression into multiple lines so that you can set a breakpoint somewhere in the middle. ■

When you start a debugging session with breakpoints set, the program executes normally without interaction until it reaches a breakpoint. In Figure 25-5, a breakpoint has been set on line 28. A debugging session is then started. Because the line that the breakpoint is on is triggered by a click on the button, the program executes normally, showing a button, until you click it. At that time, the program is halted and the debugging windows come to the front. Figure 25-5 shows the Flash Professional debugger right after the button was clicked.

FIGURE 25-5

The debugger becoming active at a breakpoint

Breakpoints are great for when a program misbehaves in a way that doesn't throw errors. If you are getting unexpected results and you have an idea where they might be originating, you can break there, investigate, and confirm your suspicion.

Stopping on Demand

Finally, you can halt execution manually, just like grabbing the bullhorn and yelling, "Cut!" This method of gaining control over debugging is more difficult to control, but you can use it to catch infrequent bugs that don't throw exceptions and manifest themselves in a way you can see. When you halt the program manually, it's rarely in a useful place, like the beginning of a frame or a timer tick. But you can take the opportunity to set new breakpoints and then continue normal execution until they are hit.

In Flash Builder, you can stop execution at the next line of source code to be run by selecting Suspend from the Run menu or the Debug view. In fdb, when the program is running normally, press Enter and confirm the prompt to break normal execution at the next line of ActionScript.

In Flash Professional, there is no way to force execution to halt immediately. However, in all three debuggers, you can add a breakpoint while the program is running, which you can exploit in much the same way to break the program when you see something you want to investigate further.

All in all, this method of halting execution is not as useful as setting breakpoints and often is used as a means to set breakpoints.

Pulling Back the Curtain

You can gain a complete understanding of your program by tightly controlling its execution and by examining the state at any step. This section teaches you how to see inside Flash Player, and the next one covers how to control execution.

Using the Variables panel in Flash Professional, the Variables view in Flash Builder, and the `print` or `p` command in fdb, you can examine the value of any variable in scope at the current line.

ActionScript developers may be used to adding `trace` statements to find out the value of a variable. Logging with `trace()` or a custom logger is still a possible debugging technique, but examining the variables interactively while execution is halted enables you to spot things you weren't looking for, gives you feedback instantly, and lets you investigate whatever you want without adding code or recompiling.

Tip

Logging a value can be more effective than inspecting variables in certain cases. Remember to always use the tool that is best suited to the task! When you have to see what range of values a variable can take, or you want to collect a large sample of possible values, it's better not to have to stop execution for every measurement. In these cases, logging the values and analyzing the results afterward may be a better technique. ■

Interpreting the Variables Panel

The Variables panel is a tree view in two columns. The left column shows the name of the variable or property, and the right column shows its value. Because types are maintained at runtime, the debugger can format these appropriately: `Strings` appear in quotes, `ints` appear as numbers and optionally with a hexadecimal interpretation, and `Booleans` appear as `true` or `false`. Complex types and

Arrays and Objects that are non-null display their type and their location in memory, and you can expand them to see, in turn, their contents.

In Figure 25-5, the Variables panel is visible. The code being executed is shown in the figure and listed in Example 25-2.

EXAMPLE 25-2 `http://actionscriptbible.com/ch25/ex2`

Using the Variables Panel

```
package {
  import flash.display.*;
  import flash.filters.BevelFilter;
  import flash.events.MouseEvent;
  public class ch25ex2 extends Sprite {
    protected var button:Sprite;
    public function ch25ex2() {
      button = new Sprite();
      button.graphics.beginFill(0xaaaaaa);
      button.graphics.drawRoundRect(-50, -10, 100, 20, 6, 6);
      button.graphics.endFill();
      button.filters = [new BevelFilter(2)];
      button.buttonMode = true;
      button.x = stage.stageWidth / 2;
      button.y = stage.stageHeight / 2;
      addChild(button);

      button.addEventListener(MouseEvent.CLICK, onButtonClick);
    }
    protected function onButtonClick(event:MouseEvent):void {
      doSomethingTricky(event.localX, event.localY);
    }
    protected function doSomethingTricky(x:Number, y:Number):void {
      var distance:Number = Math.sqrt(x * x + y * y);
      trace("Distance:", distance);
    }
  }
}
```

The debugger is halted at a breakpoint set at the line presented in bold in the preceding code. The Variables panel in Figure 25-5 displays all the variables in scope. This includes local variables bound to the arguments of a method, such as x and y; local variables created with var, such as distance; and instance variables of the class the code is executing in the scope of — in other words, everything under this. If you were to expand the view of this, you would find the instance variable button, but also all the properties of a Sprite such as scaleX and visible, because the example extends Sprite.

The variables in scope depend on where in the code you're looking, and they represent the variables' state at the moment in time you are looking at. The line that your program is about to run is marked

with an arrow, which in Figure 25-5 is superimposed over the breakpoint symbol because the program is halted exactly at the breakpoint. You can see that the `distance` variable is undefined. That's because the current line calculates `distance`, and it has not run yet. You can see the values of `x` and `y`, and you can see the type of `this`, which verifies the class that is currently in context.

You can configure the Variables panel to show all the fields of a class, including its private instance variables, implicit getter functions and derived properties, constants, and static variables, by using the Variables panel menu (the icon with the downward facing arrow on the top right corner of the panel).

Flash Builder Variables Panel and Watches

The Variables view in Flash Builder has some convenient options not present in the Flash Professional debugger. Flash Builder gives you more temporal feedback on variables: when variables change or are assigned new values, they flash red momentarily. The Flash Builder Variables view enables you to see the currently selected variable in a detail pane, which uses text wrapping so you can see long strings, XML, and other verbose variables in detail.

In addition, Flash Builder's Flash Debug perspective adds an Expressions view to the Variables view. This is similar to the Variables view, but it lets you watch just the values you are looking for, and it lets you construct simple expressions that are evaluated at the current line in the debugger. This is handy for keeping an eye on certain values or watching for a specific situation that you can write an expression to test for. You manually add and remove watch expressions from this view. For example, you could watch a deeply nested property without the need to roll down all the objects in the Variables panel by adding an expression like:

```
this.loader.content.width
```

Tip

Often, expressions you write can be evaluated only in a certain scope. Outside this scope, your watch expression displays in red as `<errors during evaluation>`. Don't let this stop you from creating the watch expressions that help you in a particular scope. ■

In the command line debugger fdb, you can add and remove watch expressions with the `display` and `undisplay` commands.

Navigating through Code

With an interactive debugger, you can drive the execution of your program. Once you break normal execution of the code, you can take over, moving and jumping ahead in your program. (Unfortunately, you can't jump back.)

Continue

You can resume normal execution of a suspended program by continuing. You will give up debugging control over the program until execution is again halted by another breakpoint or uncaught exception.

You can use a combination of continue and breakpoints to jump a long distance forward in your program. When execution is halted, you can set a new breakpoint and then click Continue to jump right to it, provided Flash Player doesn't hit another breakpoint or uncaught exception first.

You can continue a suspended program in Flash Professional by choosing Debug ➪ Continue or by clicking the green right-facing triangle button in the Debug Console panel.

In Flash Builder, the command is called Resume, and it is found in Run ➪ Resume or as a yellow bar and green triangle button, reminiscent of the Frame Advance button on a VCR, in the Debug view.

In fdb, type c or `continue` to continue a suspended program.

Continue can be useful if you have a breakpoint set on a function triggered multiple times or by user interaction, and you are interested in it only in certain cases. When the program breaks and you don't have anything to debug in this particular case, you can click Continue and wait for the code to be invoked again.

Tip

If you are using Flash Builder's Flash Debug perspective, as an alternative to setting a breakpoint and continuing multiple times when it's not relevant, you can set a breakpoint and toggle whether it is active or not using the Breakpoints view. This way, you can keep your breakpoints set but ignore them until you're ready to trigger the case you want to debug. An inactive breakpoint will not break the program when it is reached. ■

The Call Stack

As the ActionScript virtual machine executes your code, it has to shift gears all the time. Every time you call a function, the AVM must remember what it was doing, what the scope was, what all the local variables were immediately before the function call. With this information secured away, it can start with a clean slate in the new function, which may be running in a different scope (as a bound method) and with a new set of local variables. Similarly, once it reaches a `return` statement, it must grab that return value; go back to whatever it was doing before, with all the scope and local variables restored; use the returned value as the evaluation of the function call; and continue executing.

Every time the AVM enters a function, it adds a new *stack frame* onto the *call stack*. This frame represents the new environment: the scope, the local variables, and the location in the code. The new stack frame is added on the top of previous frames, which provide a trail of crumbs back to where the outermost method was invoked. When the function returns, that executing frame is complete, and it pops off the top of the stack. Finally, the AVM returns to the calling function and its stack frame, which is the new top of the stack. Therefore, the top of the stack always represents the currently executing environment.

In the following example, if you create a new A, the stack builds up as the constructor for A calls the constructor for B, and the constructor for B calls the method c(). As each function returns, however, it returns control to the line that called that function, and so on until the end of the constructor of A is reached, and the work of making a new A is over.

```
class A {
  var b:B;
  public function A() {
    //1. call stack is A::A
    b = new B();
    //6. call stack is A::A. Now return and stack is empty.
  }
}
class B {
  public function B() {
```

```
        //2. call stack is B::B, A::A
        c();
        //5. call stack is B::B, A::A
      }
    protected function c():void {
        //3. call stack is B::c, B::B, A::A
        return;
        //4. start going back up
      }
  }
```

The call stack gives context to the code that's being executed. When you view the contents of the call stack, you're looking at a *stack trace*. Although you might be looking at a line somewhere in a class, that line is executing as the result of a chain reaction of method calls, which the stack trace is documentation for.

Both the Debug Console panel in Flash and the Debug view in Flash Builder give you an interactive view of the call stack. You can use it as a stack trace, to see why the current line of code is being run. In Figure 25-5, you can see that the current method is doSomethingTricky() because it is at the top of the call stack, and it was called by the entry below it, onButtonClick().

In fdb, use bt to print a *backtrace*, another term for a stack trace.

The call stack in the graphical debuggers is interactive: if you click a stack frame below the current one (which is, again, always on top), the code view, current line, and variables view refresh to show you the state of the suspended stack frame. You can see the code that called the method you're currently debugging and the state of the scope and locals in that frame when it invoked the function.

Step Into

You can move a line forward in your program with ease. When you do this repeatedly, you are *stepping through* your code. In general, a step takes you forward one line of the program. But even this definition has some ambiguity, and there are three kinds of steps you can take to step forward. Step Into takes you to the next line of code to execute, drilling into any calls the current line might make.

For example, if this line of code were up next:

```
        var name:String = getName(); //<---
        name = "Mr. " + name;
```

Step Into would transport you to the first line of the getName() function, rather than to the line that adds "Mr."

Step Into drills its way into accessor functions, even if they are implicit. So when you see an innocuous assignment such as the following:

```
    color = child.color;
```

you may end up stepping first into the implicit getter function for color (on child) and then into the implicit setter function for color on this. A single line like this, when executing, can end up jumping to many different parts of a program before finishing. Step Into follows every line that the AVM executes without sparing you details.

Caution

While stepping into, the AVM2 debugger does not enter Flash Player API calls. These are part of the Flash Player software and are not written in ActionScript, nor do their internals become part of your program. The debugger steps through lines of code that are compiled into your program from ActionScript or SWCs. However, the Flex framework is written in ActionScript, and the AVM2 debugger does attempt to step into the framework's internals if you step into a Flex API call. ∎

Stepping through code while inspecting variables gives you a perfect view of how your program actually works (or fails to work). It is the ultimate in virtual machine voyeurism.

Step Over

Step Into might provide you with more excruciating detail than you need. If you find yourself impatiently clicking Step Into over and over, you should either jump ahead to another breakpoint or use Step Over. Step Over lets you jump to the next line in the current scope, if it exists, jumping over any function calls that exist in the current line. These calls are performed and completed, but you don't have to see them step by step; just as when you continue until the next breakpoint, all code in between is executed normally.

Use Step Over when you're interested in the behavior of a certain block of code or algorithm and want to track it without shifting perspective into other stack frames. Step Over may be the most useful kind of step.

Step Out

Step Out (in Flash Professional) and Step Return (in Flash Builder) are used to jump out of the current stack frame. They execute everything in the current function until the function returns, and they leave you at the previous stack frame, where the function was called from.

Step Out is useful when you accidentally step into a function, or when, in looking at code in the debugger, you develop a suspicion that the problem lies in a calling function rather than the current one. You can somewhat preview a Step Out operation by examining the stack frame that you would return to by actually stepping out, but that won't do anything to actually move the current line or step forward in the execution of the code.

Debugging a Simple Example

In this example, I tried to make a rocket ship point toward your mouse cursor. But I left in an error that you can use the debugger to help find. First, the code:

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.Point;
  public class RocketShipExample extends Sprite {
    protected var ship:MovieClip;
    public function RocketShipExample() {
      ship = getChildByName("rocketshipMovieClip") as MovieClip;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
```

```
            var p:Point = new Point(stage.mouseX, stage.mouseY);
            p.subtract(ship.localToGlobal(new Point()));
            var radians:Number = Math.atan2(p.y, p.x);
            ship.rotation = radians * 180 / Math.PI;
        }
    }
}
```

This class was set as the document class of a new Flash file. I drew a rocket ship, made it a movie clip symbol with the registration point at the center, and set its instance name on the stage to `rocketshipMovieClip`.

On every frame, the class changes the `rotation` of the `ship` so that its nose points toward the mouse cursor. It does this by constructing a vector that points from the center of the ship (the `MovieClip`'s registration point) to the current position of the mouse. Then it takes the arctangent of this vector to get its angle and assigns that rotation to the ship, converting from radians to degrees along the way.

Or, that's the way it should work. But when you run this, the rocket ship only seems to point between right and down (0 to 90 degrees). There's something wrong, but no errors are thrown, so it's up to you to figure out where the error is. The angle that's produced is incorrect, so you know that the error must be somewhere in the lines that produce that angle (lines 15–18, as shown in Figure 25-6).

FIGURE 25-6

Debugging the rocket ship example

You'll figure out what is going wrong with the angle calculation by looking at the variables involved. Set a breakpoint at the beginning of the calculation, line 15. When you start debugging, the program immediately breaks before you can see what's going on. This might be a good opportunity to add a breakpoint at runtime, so you can try removing the breakpoint, seeing how the rocket ship reacts to the mouse, and adding the breakpoint during the debug session when you're ready to dive in. Because code runs every frame, adding a breakpoint inside `onEnterFrame()` while the program is running suspends execution on the next frame.

The first two lines create a `Point` object that defines the vector between the rocket ship and the mouse, or, to think of it another way, the location of the mouse with the origin placed at the center of the ship. First, the point `p` is set to the absolute position of the mouse. To see this assignment, you can step into or step over after the breakpoint. This executes the line you were at (the breakpoint, line 15). In the Variables panel, `p` turns from `null` into a `flash.geom.Point`. Use the Variables panel to examine the contents of this object by clicking the plus/minus box. The `Point` contains properties `x`, `y`, and `length`. In Figure 25-6, you can see that `p.x` is 17 and `p.y` is 84.

The next line shifts that point so that it's relative to the position of the rocket ship rather than absolute. It gets a new `Point`, which is initialized to (0, 0), and converts that location on the rocket ship to a global location. By converting the origin of the rocket ship to the global coordinate space, you find the absolute location of the ship. Then that location is subtracted from the position of the mouse to find the position of the mouse relative to the ship. Again, Step In or Step Over to execute the line. Note that because the inner function calls in this line are calls to the Flash Player API, Step Into does not drill into them. After you execute this line, keeping your eye on the Variables panel, you should notice that the properties of `p` don't change. This means that either the rocket ship is at the origin of the global coordinate space (because subtracting (0, 0) wouldn't do anything) or there is an error in this line. If you want to eliminate the possibility of the first case, you can stop debugging to create an intermediate variable that you can then monitor, as discussed in the next section. However, if you check that line more carefully with the knowledge that `p` doesn't change, you should be able to realize the error in this code.

Calling `subtract()` on `p` doesn't change the value of `p`. Why? Because `Point::subtract()` is a nondestructive method. It returns the result of the subtraction rather than performing it on itself. Mixing up destructive methods with nondestructive methods is a common programming slip-up, especially when the method name doesn't hint at whether it is destructive or not.

To fix the example, simply change line 16 to read as follows:

```
p = p.subtract(ship.localToGlobal(new Point()));
```

Interactive debugging is an investigative science. You have to use the tools in the debugger together to collect facts or run tests that confirm or disprove hypotheses. The most important tool in debugging is your brain. The debugger doesn't debug your programs — you do.

Using the Debugger Effectively

I've covered all the basic techniques you can employ to debug programs in ActionScript 3.0. As you use the debugger, you can develop your own combinations of techniques that help you figure out different kinds of problems. Here are a few techniques that can help you get started.

When you encounter an uncaught exception, the call stack is especially useful in figuring out why it happened. In these cases, you have no control over where the program breaks, so rather than go back in time, which you can't do, you can go up the call stack, examining the Variables view in previous stack frames to, at a minimum, build up a case that can replicate the error or, at best, determine the problem right off the bat.

You should use the Variables panel or Variables view liberally to not just see what variables are at a certain line but watch them as the code executes. However, you can also use it to assign new values to variables at runtime. You can use this to see how your program will deal with invalid input or exceptional cases, and you can use it to instigate rare errors manually, if you know what situation leads to that kind of error.

When you have complex expressions that you can't step into, you might consider — at least temporarily — splitting the expression into intermediate variables. This way, you can debug the expression piece by piece when you could not otherwise. For example, you might break up this:

```
return (Math.sqrt(x * x + y * y));
```

into this:

```
var x2:Number = x * x;
var y2:Number = y * y;
var sum:Number = x2 + y2;
var ret:Number = Math.sqrt(sum);
return ret;
```

This way, you can use the Variables panel to examine all the constituents of the expression one by one.

A similar technique is to use the Variables panel to try out several options for evaluating an expression if things aren't working right one way. Again, create several local variables to hold the results of the expression evaluated multiple ways. You can then set a breakpoint after they're evaluated and check them out in the Variables panel. This is a lot better than tracing out their values because you can dig into their references if they are complex objects.

At the top of a function, you might notice that all the local variables are already populated in the Variables view, but with unassigned values (`null` for objects, `NaN` for `Number`s, and so on), even if you don't declare these local variables until later. This is because the ActionScript 3.0 compiler uses variable hoisting: all the declarations for local variables are moved to the top of the function even if they don't get an assignment until later. This fact shouldn't affect your program, but you will be reminded of it by the compiler if, for example, you create a variable inside a loop block with the same name as one that's declared elsewhere in the method.

You can use Continue as a kind of code stepping for loops. Set a breakpoint at the first line of the body of a loop, and click Continue to step to the next iteration of the loop body.

Given an interactive debugger, some techniques for utilizing it, and a forensic approach, you should be able to fix bugs in your code quickly, effectively, and without resorting to trial and error.

Summary

- The AVM2 debugger is an interactive tool that helps you fix problems in your program.
- There is a version of the AVM2 debugger for each product that uses ActionScript 3.0.
- The debugger must be launched with a SWF compiled for debugging and using a debugger version of Flash Player.
- IDEs such as Flash Professional, Flash Builder, and Flex Builder let you launch debugging sessions in one click.
- The program executes normally until the debugger takes over.
- The debugger can take over at an uncaught exception or a breakpoint.
- You can use the debugger to examine, and even change, properties of any variable in scope.

521

- You can follow any reference in the debugger to reach nested or associated objects.

- You can watch the variables change as you move through the program.

- The call stack allows you to see what called the current code.

- The call stack enables you to switch perspective through all the lines that called methods to get to the current line.

- You can examine local variables at other depths of the call stack.

- You can move through the program with Step Into, Step Over, Step Out/Return, and Continue.

- The debugger is a set of tools, but the real debugger is you. Set up experiments and test cases that can help you get to the bottom of a bug.

Making Your Application Fault-Tolerant

You can call it many things: fault-tolerant, bulletproof, robust. However you describe it, an application that doesn't shatter when things go wrong is a joy to use. These kinds of applications behave well not through luck or coding practices but because an extensive error-handling strategy was devised and applied. This chapter introduces the kinds of errors that can occur and general strategies for handling them.

Developing a Strategy

Why does your application need an error-handling strategy? Despite the best efforts of the smartest people, things always, always, *always* go wrong. If you do nothing to recover from errors, your program can appear to malfunction to the end user. In Chapter 24, "Errors and Exceptions," you learned that uncaught exceptions in the release versions of the Flash Player terminate all code executing on the frame. Although this might not sound so bad — at least the program keeps going on the next frame, right? — terminating all code for a frame can be disastrous, skipping vital code and leaving the internals of your program in an invalid state. Imagine if one day, without warning, you were transported forward in time a full year. You would find your taxes have not been paid; you haven't showed up to work; and you haven't answered your spouse's calls. Things would probably not be so great for you. Just like life, most programs have to finish things up in an expected way before moving forward, so you can't rely on the default behavior of uncaught exceptions.

Handling failures is not as simple a topic as it might seem. Especially for large applications, it's important that you develop (and even document) a strategy for handling failures. To do this, you need to determine what kinds of failures you should handle, how you should handle them, and what kind of information to collect and display about them.

Determining What Errors to Handle

Catching exceptions is only the beginning of an error-handling strategy. Any kind of input that you are expecting from outside your program, especially from the internet or a user, should be validated. You should be prepared for any request from a network to fail, including both multimedia content and server requests. Not only could the resource you are looking for have been moved, but it's possible that the server hosting it is down or unavailable, or the user's computer's connection is terminated or disrupted.

If your project has a server-side component, code on the server might fail. When this happens, the request might complete successfully from a networking perspective, but the result might not be a valid response. For example, you might make a request expecting an XML file to be served in response but receive an error from the web server or the script's interpreter. If you go to `http://example.com/dosomething.php?make=measammich` and expect something like this:

```
<?xml version="1.0"?>
<response>
    <banhmi/>
</response>
```

there may be a chance that an error in the server code will result in a response that isn't valid XML:

```
<b>Notice</b>:  ARGGH i died! in
<b>/usr/local/share/apache/www/example.com/dosomething.php</b>
on line <b>1</b>
```

If you have any control over the server-side component of a client-server application, you should ensure that server-side failures are communicated to your program in a way that you can capture that information. You need to be able to determine that an error happened on the server side, and you have to prevent your program from interpreting the error message as it would normal data.

Even if you include measures that validate security concerns, it's possible that your requests might still fail if the user specifically denies them. You should handle denied requests, even when they should be accepted by default. For example, requests to store information locally with a `SharedObject` (see Chapter 29, "Sharing Data Locally with SharedObject") or capture audio from the user's microphone (see Chapter 33, "Capturing Sound and Video") may require permission from the user. Users can also use software or their browser preferences to deny storing cookies or opening pop-up windows.

It's not possible to list all potential causes for error that you should handle, but here is a partial list:

- Errors loading XML/text
- Strings/XML not formatted as you expect
- Errors loading images/video/sound/SWFs
- Events coming from unexpected targets
- Functions that return error codes or `null`
- Functions that raise exceptions
- Server-side errors
- Timeouts in network requests
- Security sandbox violations
- State of the display list not as expected

In addition, keep your eyes open to the kinds of errors you get as you are actively developing your program. An error that occurs to you once in a freak accident can point you to a weakness in your error-handling strategy.

Categorizing Failures
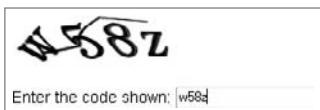
After identifying the kinds of errors that might occur in your program, you can determine how to react to them. For each potential error, there might be steps you can take to alleviate the problem. For some errors, however, there may be no way to recover. In determining how to handle an error, you must first decide if it is possible to recover from it. These kinds of decisions affect how the application works for the end user, and they should be made with the end user's experience in mind.

For example, if an image fails to load, you can draw a "broken image" icon or placeholder image in its place. But think about how it will make your program behave. In some cases, drawing a place-holder image might not be appropriate, for example if the image is a CAPTCHA image. CAPTCHA images are simple queries that theoretically prove that a human user is interacting with the computer by requiring some trivial task that is difficult for a computer, typically recognizing a distorted word, as demonstrated in Figure 26-1. If this image fails to load, showing a replacement image won't be an acceptable solution, as Figure 26-2 makes evident. This example illustrates the need to react to errors based on the user's experience. A good strategy in one case can be unacceptable in another case.
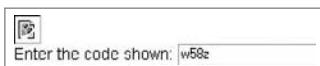
FIGURE 26-1

A program using a CAPTCHA image

FIGURE 26-2

A program using an error-handling strategy for images that fail to load

In general, your intent in handling an error should depend on the severity of the failure. If the error is nonfatal, your intent should be to return the program to a state in which everything else can continue normally. If the error is fatal, your intent should be to try again, or gracefully terminate the program.

Returning the program to a normal state can be difficult, especially if other areas of the program depend on something that failed. You must also ensure that even if a runtime error disrupts the flow of control of your application, you can resume and perform necessary cleanup code. In other words, make sure you utilize the `catch` and `finally` blocks when handling runtime errors.

In addition to your primary goal of returning the program to normalcy, two actions are useful to any error-handling strategy: logging errors and reporting failures to the user.

Logging Errors

Why should you log errors? When developing an application, these logs can be vital to your own development and bug-fixing process. Once the application is deployed, the logs can help you collect and fix bugs that appear in the wild. For instance, if the person who paid for the software you developed has some problem with it on her computer at home, the hope is that you will still be able to investigate the causes of that failure.

Logging, although it technically doesn't help you handle an error once the error occurs, is an essential part of any long-term error-handling strategy. Without logging, you won't be able to tell the difference between a program that is handling lots of faults correctly and a program that is having no trouble at all. Error logs help you start identifying problems without a debugger (remember that a debugger only works with SWFs compiled for debugging, which the public versions of your application probably won't be), and they let you capture error information in a way that you can control and retain.

Logging can be tricky because there are many ways to get information out of a running SWF. Your approach can greatly depend on how your program is deployed, as well. Therefore, the implementation details of different logging methods are beyond the scope of this book. Some ways you can log information, in brief, are

- Using `trace()` output — This method can be viewed only if you have the debugging version of Flash Player and have it configured correctly for logging. This approach takes no effort to write code for, but it can be complicated to set up properly, especially if you need to capture logs from a SWF running in a browser. Mark Walters explains how in his post at `http://bit.ly/capture-trace`.

- To a separate application through sockets or a `LocalConnection` — This approach doesn't require special setup of the Flash Player but does require an external application to capture the logs. Several third-party products provide the external log viewer and the framework to log to it. Popular loggers (that often provide further debugging features) include SOS (`http://fdt.powerflasher.de/developer-tools/sosmax/home/`), XRay (`http://osflash.org/xray`), and Arthropod (`http://arthropod.stopp.se/`).

- To the browser or enclosing web page using JavaScript — One example of this is logging to FireBug, a debugging extension to the Firefox browser. This approach requires that specific browser and extension, but it is easy to set up. One implementation of this idea is hosted at `http://bit.ly/logger-marumushi`, another at `http://bit.ly/logger-grumblecode`.

- To a file, if the program is running on the desktop with Adobe AIR — One approach is described at `http://dispatchevent.org/roger/logging-to-a-file-in-apollo`.

- To a server — You should be careful of both network overhead and privacy concerns if you choose this method. You should let the user know exactly what is being transferred and avoid logging personal or sensitive information at any cost.

In all these cases, you should log information that will help you debug problems if they arise. A message in the log that says, "Something went wrong!" is useless. Log a description of what went wrong and where. The following snippet shows specific messages depending on the kind of error that happened:

```
public function displayText():void {
  try {
    var tf:TextField = TextField(getChildAt(0));
    tf.htmlText = "Hello World!";
  } catch (error:RangeError) {
    Console.log("displayText(): No display child found at position 0!");
```

```
    } catch (error:TypeError) {
      Console.log("displayText(): Display child at position 0 \
is not a TextField!");
    }
  }
```

All the preceding logging methods require you to write slightly different code, so in place of `Console.log()` you might see `trace()` or `Log.getLogger().log()` or something entirely different. The concept is always the same, however: get a message out of the program without notifying the user.

Along with a message, it is typical to log the severity of the error. The canonical error severities are:

- `info` — Not an error, just information. You should remove most of these before deploying your program for general use.

- `warn` — A warning. Something is atypical and might cause an error later, or an easily recoverable error was handled.

- `error` — Some kind of error. Should be recoverable.

- `fatal` — An error that can't be recovered from. The program must terminate.

A severity is sent along with a logging message in some way similar to this:

```
Log.getLogger("com.actionscriptbible.Shell").fatal("Can't load main SWF!");
```

ActionScript 3.0 doesn't specify how logging should work or the syntax for logging errors. However, if you are using Flex, it has a good logging framework that you should employ. The preceding example shows the syntax used in the Flex framework logging system. The class name `com.actionscriptbible.Shell` is used here as a logging category to help you identify where the error came from.

Whichever way you choose to log errors, having a record of those errors will benefit you.

Messaging the User

An important component of handling errors — both ones that can be recovered from and especially fatal ones — is messaging the user. In some cases this can be implied, such as using a "broken image" icon. In other cases you can reuse a general error dialog, like the one shown in Figure 26-3, to communicate explicitly.

FIGURE 26-3

A reusable message dialog box

If you are doing something that might take a while, you should definitely warn the user. For example, if a server call fails to return, you could wait a moment and try it again. If this is going to take more than a second or two, it might be good to let the user know that there is some network trouble.

If the server fails outright when the user requests some action, the expectation that the request was fulfilled is not just your program's but the user's as well. You should let the user know if a request she initiated was not completed normally. For example, if she clicks the Save button but the server can't save the file, you should be a pal and tell her that her document wasn't actually saved. Otherwise, she might close your program, lose her changes, and will probably be very unhappy when she finds out much later what happened, filing a complaint with the Ministry of Flash. They will then find out you read this book and undoubtedly forward the complaint to me, and I will be very, very, disappointed with you.

Of course, you want to limit the feedback you give users. We all know how irritating it can be to be assaulted with dozens of confirmation dialog boxes that you don't care about. If the user doesn't need to know that you recovered from an error successfully, just log it. Remember, I get those complaints, and I read every one.

In the case of a completely fatal error, the best thing you can do is inform the user and, depending on who the audience is, consider providing an explanation. We've all had the experience of staring at a stalled, or blank, or messed-up screen, wondering if we should just keep waiting or give up and reboot. Not knowing makes the experience even more harrowing. Although you should do everything in your power to avoid the situation in the first place, making a complete failure into an elegant, apologetic event can parlay a user's frustration into some measure of appreciation.

Degrading Styles: An Example

Example 26-1 shows a text field that tries to display a message with some styling. Because the style isn't as important as the message, failing to load the style sheet isn't treated as fatal. In all failure cases, a bare-bones style, which is compiled into the program, is used. Additionally, a timer stops the stylesheet from loading if it takes more than 2 seconds.

EXAMPLE 26-1 `http://actionscriptbible.com/ch26/ex1`

Gracefully Handling Load Failure

```
package {
  import com.actionscriptbible.Example;
  import flash.events.*;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.text.StyleSheet;
  import flash.text.TextField;
  import flash.utils.Timer;

  public class ch26ex1 extends Example {
    protected const STYLESHEET_LOCATION:String =
      "http://actionscriptbible.com/files/example.css";
    protected const BASIC_STYLE:String =
      "a {text-decoration: underline; color: #0000ff}";
    protected const TIMEOUT:int = 2000;
    protected var timer:Timer;
    protected var loader:URLLoader;
    protected var tf:TextField;

    public function ch26ex1() {
      makeTextField();
```

```
    loadStyleSheet();
    trace("\n\nLogging --------------");
  }
  protected function makeTextField():void {
    tf = new TextField();
    tf.width = stage.stageWidth;
    tf.height = stage.stageHeight;
    tf.multiline = true;
    tf.wordWrap = true;
    addChild(tf);
  }
  protected function loadStyleSheet():void {
    loader = new URLLoader();
    loader.addEventListener(IOErrorEvent.IO_ERROR, onLoadError);
    loader.addEventListener(SecurityErrorEvent.SECURITY_ERROR,onLoadError);
    loader.addEventListener(Event.COMPLETE, onLoadSuccess);

    loader.load(new URLRequest(STYLESHEET_LOCATION));

    timer = new Timer(TIMEOUT, 1);
    timer.addEventListener(TimerEvent.TIMER, onTimeout);
    timer.start();
  }
  protected function onLoadError(event:Event):void {
    removeEventListeners();
    trace("Error loading CSS:", event.type, ". Using basicstyle.");
    setDefaultStyle();
  }
  protected function onLoadSuccess(event:Event):void {
    removeEventListeners();
    try {
      var cssText:String = loader.data;
      var styleSheet:StyleSheet = new StyleSheet();
      styleSheet.parseCSS(cssText);
      setStyle(styleSheet);
      trace("CSS set successfully");
    } catch (error:Error) {
      trace("Error parsing CSS. Using basic style.");
      setDefaultStyle();
    }
  }
  protected function onTimeout(event:TimerEvent):void {
    trace("CSS loading timed out. Using basic style.");
    loader.close();
    removeEventListeners();
    setDefaultStyle();
  }
  protected function removeEventListeners():void {
    timer.stop();
    timer.removeEventListener(TimerEvent.TIMER, onTimeout);
    loader.removeEventListener(IOErrorEvent.IO_ERROR, onLoadError);
```

529

```
    loader.removeEventListener(SecurityErrorEvent.SECURITY_ERROR,onLoadError);
    loader.removeEventListener(Event.COMPLETE, onLoadSuccess);
  }
  protected function setDefaultStyle():void {
    var styleSheet:StyleSheet = new StyleSheet();
    styleSheet.parseCSS(BASIC_STYLE);
    setStyle(styleSheet);
  }
  protected function setStyle(styleSheet:StyleSheet):void {
    tf.styleSheet = styleSheet;
    tf.htmlText = '<p>Welcome to the <span class="title">AS3Bible</span>! ' +
                  'Go <a href="http://actionscriptbible.com/">here</a>.</p>';
  }
  }
}
```

The example responds to all errors that might arise when loading the CSS file from the internet, in `onLoadError()`. If there's an error parsing the CSS for any reason, you catch the errors in the `catch` block of `onLoadSuccess()`. A timer is also used to stop waiting for a long load (perhaps a bit contrived, as a very long delay will eventually trigger a load error on its own), calling `onTimeout()` if the program is tired of waiting. All of these log a message with `trace()`, and all of them fall back on a recovery strategy — using a built-in stylesheet, set in `setDefaultStyle()`.

Another approach to this problem is to add the text into the text field at the beginning and redraw the text with the appropriate style after the stylesheet loads successfully. This way, the user can see the content without delay. On the other hand, it might produce a visible flicker as the text is redrawn. These kinds of considerations are at the core of an error-handling strategy.

Summary

- It's necessary to come up with a comprehensive error-handling strategy.
- Error handling isn't just about `try/catch` blocks.
- You should handle errors that occur from a variety of sources, including from any remote or unverified source.
- Some errors you can recover from; others are fatal.
- Think of errors from the user's standpoint: how does the program behave?
- When you handle an error, your job is to return the program to a normal state and make sure that dependent code will be okay.
- When errors occur, log them.
- Logging can be done many different ways, each with its pros and cons, depending on the environment your program executes in.
- When errors affect the user, let the user know with some kind of feedback.
- When errors are fatal, let the user know.

Part VI

External Data

Networking Basics and Flash Player Security

A time will come where you, like Leif Eriksson, will yearn to leave the fjords of Iceland, which you're accustomed to, and explore the great unknown across the Atlantic. Now, replace "fjords of Iceland" with "the contents of memory" and "the Atlantic" with "the internet," and you've got something to talk about. The experience of crossing the sea differs for each who crosses it, and so varies the manner in which you can communicate with the vastness of the internet. Besides containing an incomprehensible variety of data in a comparably broad multitude of formats, the internet has innumerable services you can communicate in both directions with. In this and the following chapters, I'll dive into the many ways Flash Player can make use of the internet. In this chapter, I'll start with some of the most useful and simple ways to do so. I'll present an overview of the most common and best-supported protocol in Flash Player for networking, HTTP. You'll use ActionScript 3.0 to open a web page, load graphics and SWFs, and load files. Finally, I'll look at Flash Player's security model, which you must be aware of when developing networking-enabled applications.

HTTP in Brief

The internet is kept alive by the furious and continuous exchange of information. This exchange takes place in a variety of protocols. These application-layer protocols are agreements between two computers connected to the internet on how they should structure their communication. Some examples of application-layer protocols are FTP, Bittorrent, DNS, IMAP, POP3, and SMTP. The whole of the World Wide Web, however, is built on one application layer protocol: the HyperText Transfer Protocol (HTTP).

Note

The internet itself relies on a whole stack of lower and lower-level protocols. Each layer of protocol builds on the ones below it, providing more structure. You can learn more about the seven-layer model of networking protocols, or OSI model, in a networking text or online (`http://en.wikipedia.org/wiki/OSI_model`). In addition, although not part of the burrito-like seven-layer model, there are protocols on protocols that live on top of HTTP. I'll touch on some of these higher-level protocols in Chapter 28, "Communicating with Remote Services." ∎

Because Flash content so often lives on the web, HTTP is ingrained in Flash Player, and you'll soon be using it to load and send data, so let's take a brief excursion into how it works. I'll try not to bore you to tears with detail but provide just what you need. If you have a grip on HTTP, feel free to skip this section.

There are two participants in any HTTP exchange. I'll call them the client and the server. Let's consider a simple case: you use your web browser to display `http://actionscriptbible.com`. Now, your computer is the client and some big, fat, rack-mounted, air-conditioned computer in Los Angeles is the server. Actually, I've never seen the server, but it doesn't actually matter what it is or where it is — that's the whole point of the internet.

Back to the example. Every HTTP exchange has two acts: a *request* and a *response*. Simple enough. The client asks for something, and the server gives it back. Your browser opens a connection to the server hosting `actionscriptbible.com` (using the DNS protocol to figure out that server's IP address, if you care) and sends it a request for /. If you were loading the page `http://actionscriptbible.com/crossdomain.xml`, it would request `/crossdomain.xml`. The server then tries to find the resource you requested, and sends you back what it finds. This step can have lots of interpretation going on in the background: a script can be run by the server, it can alias your request to a different location, it can fail to find the resource you're requesting, and so on. This is the basic pattern. You send a request, and the server sends a response.

Let's look a little bit closer at the structure of a request and a response. Although you won't often have to write these by hand, it's nice to know that these take place in plaintext and that the HTTP protocol is pretty readable. Let's take a quick peek at what the example transaction looks like.

First the client sends a request to the server:

```
GET / HTTP/1.1
Host: actionscriptbible.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.13)
Gecko/2009073022 Firefox/3.0.13 (.NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

Then the server sends the response to the client:

```
HTTP/1.1 200 OK
Date: Fri, 14 Aug 2009 21:06:17 GMT
Server: Apache
```

```
Last-Modified: Mon, 08 Jun 2009 17:38:10 GMT
Content-Encoding: gzip
Content-Length: 39
Content-Type: text/html

<html>...
```

This is followed by the contents of the HTML file `index.html`. These examples are real world and contain more information than you need to worry about. For example, my browser has signaled that the server can send data compressed, and the server has replied that the contents are indeed compressed. My browser has also sent along information about what kind of browser it is and what languages I prefer to read in. These are incidental details! The most important lines in the request and response are the first lines of each, which I've bolded.

First let's look at the request: `GET /`. Requests can be for a few different kinds of actions. In this case you've performed the `GET` method, which as you might guess, asks the server to get the resource. The second argument, `/`, specifies the resource. The browser has also been so kind as to notify the server that the request is written in version 1.1 of the HTTP standard.

Note
Although current web servers have quite a limited vocabulary, using the HTTP request as a meaningful verb in client-server applications is a tenet of the RESTful ideology. For further reading, I recommend the article by Ryan Tomayko at `http://tomayko.com/writings/rest-to-my-wife`. ∎

Now the response: `200 OK`. Whether the resource is found or not, whether the requested action could be performed or whether it triggered an error, the server sends a response. The `200 OK` is the *response code*, which indicates what happened when the server tried to carry out your request. It has a numeric code and a human-readable form. For example, if you `GET` a resource that doesn't exist on the server, it sends a response with the response code `404 Not Found`. In this example, the server has interpreted the request for `/`, finding an `index.html` file in the root web directory associated with `actionscriptbible.com` and returning the contents thereof.

You'll notice that in both the request and response there are a series of lines with the form `Some-Key: value`. These are the HTTP headers, and they are used to convey both meta-information — information about the HTTP request itself — and other data. In an HTTP request or response, the end of the headers and the start of the data contained in the response is indicated by a double newline; that is, a blank line is always between the last header and the start of the response data. One of the really important headers is `Content-Type`. This header tells the recipient of the message exactly what kind of data appears in the body. Likewise, the `Content-Length` header tells the recipient how big said data is.

Loading something like a web site is usually more than a single HTTP exchange. In most cases, the HTML file that the server retrieves for you includes references to all kinds of external files that must also be fetched, like a CSS file, several image files, maybe some JavaScript files, and sometimes even SWF files! Each one of these files is then loaded subsequently by the browser to get and render the web site in all its glory.

This should be enough to get you started with some simple networking in Flash Player. Much like a web browser, Flash Player can handle making and sending HTTP requests and interpreting HTTP responses by itself. The more you dig into the networking API, the more you'll interact with the HTTP protocol.

535

Tip

You can examine HTTP requests and responses by using a debugging proxy. This is extremely useful when debugging networking code in Flash Player. There are many proxies out there. I recommend Charles, an excellent cross-platform debugging proxy (`http://www.charlesproxy.com/`). ■

Introducing URLRequest

In Flash Player, even the simplest HTTP request is mediated by a `URLRequest` object. For every HTTP request you make, an instance of the `URLRequest` class controls how Flash Player constructs the request. Flash Player takes care of creating the actual text of the request; `URLRequest` provides a convenient interface.

The simplest request specifies a URL. You can either specify the URL in the constructor of the `URLRequest` object or using its `url` property.

```
var request:URLRequest = new URLRequest("http://actionscriptbible.com/");

var request:URLRequest = new URLRequest();
request.url = "http://actionscriptbible.com/";
```

Here you create a request for `http://actionscriptbible.com/` using both methods.

Using `URLRequest`, you can control all those things you learned about in the previous section, including the HTTP method and the headers. I'll detail these features as you need them, but because all HTTP networking methods use `URLRequests`, an early introduction is necessary.

Navigating to a Web Page

You can use a `URLRequest` to specify a URL that Flash Player should launch in your browser, just as if you've clicked a link, using the package-scoped function `flash.net.navigateToUrl()`. This is a public function that takes two parameters. The first is the `URLRequest` that contains the URL you want to navigate to, and the second is the window that you want to open the link in. The `navigateToURL()` function takes the following parameters:

- `request` — A `URLRequest` object that specifies the URL to navigate to.
- `window` — An optional `String` parameter specifying the browser window or frame to display the URL into. Special values are the same as in HTML: `_self`, meaning the current window; `_blank`, meaning a new window; `_parent`, which means the parent frame if your page is using frames; or `_top`, meaning the top-level frame in the window. The default is `_blank`.

If you're looking at a SWF embedded in a web page and you want it to replace the page with a new page, you would use the following code:

```
navigateToURL(new URLRequest("http://actionscriptbible.com/"), "_self");
```

This is frequently done in response to a click on a display object.

This function, like all networking behavior, is subject to restrictions by Flash Player's security policy. In this case, how the SWF is embedded in an HTML page can affect whether the navigation is

allowed. The `allowScriptAccess` embedding parameter has this effect. More on this in Chapter 43, "Interfacing with JavaScript."

Caution

Don't forget to import the function before use, as it is defined in the `flash.net` package.

Pop-up blockers in a browser can prevent new windows from being launched from inside Flash Player. You might want to instead try to launch the pop-up with JavaScript (using `ExternalInterface` as described in Chapter 43, as JavaScript can at least return a value telling you whether the pop-up was launched successfully. ■

Loader

Flash Player can display graphics in three ways: you can use ActionScript (and sometimes Pixel Bender) code to draw graphics on the screen (see Part VIII, "Graphics Programming and Animation"), you can use embedding to associate graphics with `Class` references (see Chapter 16, "Working with DisplayObjects in Flash Professional"), or you can use Flash Player's networking capabilities to load graphics files from the internet. The `Loader` class provides an easy-to-use interface for including the contents of external files in your Flash application.

Note

Choosing whether to embed graphics or load them externally (or even draw them programmatically when you can) is not always easy. The overall strategy you pick for including graphical content in your application can have a big impact on how you load and structure it. Because dynamically loaded graphics are easy to manipulate and there are multiple free libraries to assist in managing large external libraries, I prefer to externalize all assets in ActionScript projects. ■

The `Loader` class is a subclass of `DisplayObject`. Not only does it load in external graphics, it displays them as well. This makes it really easy to get graphics onto the screen. You just create a `Loader` instance, tell it where to request the file from, and add it into your display list like any other display object. Even more, it inherits from `InteractiveObject`, so you can use loaded graphics interactively, receiving mouse clicks and so on.

Example 27-1 shows just how easy it is to put an image on the screen using `Loader`.

EXAMPLE 27-1 `http://actionscriptbible.com/ch27/ex1`

Displaying an Image with Loader

```
package {
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.net.URLRequest;

  public class ch27ex1 extends Sprite {
    public function ch27ex1() {
      var loader:Loader = new Loader();
      //image courtesy NASA, Blue Marble Project
```

continued

```
    var url:String = "http://actionscriptbible.com/files/bluemarble.jpg";
    loader.load(new URLRequest(url));
    addChild(loader);
    }
  }
}
```

Here you can see that only three steps are required: create a `Loader` object, call its `load()` method with a valid `URLRequest`, and add it to a display list. That's it! `Loader` does all of its own loading and drawing. You can just "set it and forget it."

Graphics File Formats

One of the best things about `Loader` being so self-sufficient is that it knows how to actually decode graphics files. Because you don't have to think about it, it's easy to overlook this step that `Loader` handles. It will load all sorts of things you throw at it:

- PNGs, including 32-bit PNGs with a full 8-bit alpha channel
- GIFs, including transparent GIFs (but not animated)
- JPEGs, including progressive JPEGs
- SWFs

If you want to display graphics in other formats, Flash Player provides you all the tools you need to do it yourself (using `BitmapData`, see Chapter 36, "Programming Bitmap Graphics"), if you can tell Flash Player how to convert the binary data in the file to the pixels on the screen. One example is the AS3 GIF Player library by Thibault Imbert, available at `http://as3gif.googlecode.com/`, that lets you display animated GIFs.

The point I'm making here is that `Loader` is a simple interface to some of the built-in file decoding facilities of Flash Player, which is part of the reason that it makes displaying graphics so simple. In fact, you can utilize just the decoding abilities of `Loader` if you want, by passing in binary data to `Loader`'s `loadBytes()` method. I'll cover that in the section "Using Loader to Interpret Files in Memory."

Accessing Information about the Loading Process

The `Loader` class loads, decodes, and displays images. It also compartmentalizes these different responsibilities and gives you deeper access to them by exposing some of its properties. For instance, in Example 27-1 you loaded and displayed an image in the simplest way possible. You exercised no control over how the image was loaded, so it appeared when it was fully loaded with no intervention from you. You didn't care how big the image file was, how long it took to load, or even if the load process failed. But `Loader` does provide information on the loading process should you need it.

A `Loader` exposes instances of the `LoaderInfo` class, which provide all kinds of information about the loading process and data about the loaded images. In addition, the `LoaderInfo` object dispatches events related to the loading process. Remember that this relationship is called composition.

The `LoaderInfo` class has one specific responsibility, and the `Loader` class owns and relies on a `LoaderInfo` to do that part of its job.

When loading external images, the `LoaderInfo` object is commonly used to

- Determine how large the image being loaded is.
- Calculate the progress of a load operation.
- Notify other objects that the file is done loading.
- Notify other objects that the file failed to load.
- Retrieve the original dimensions of the loaded image.
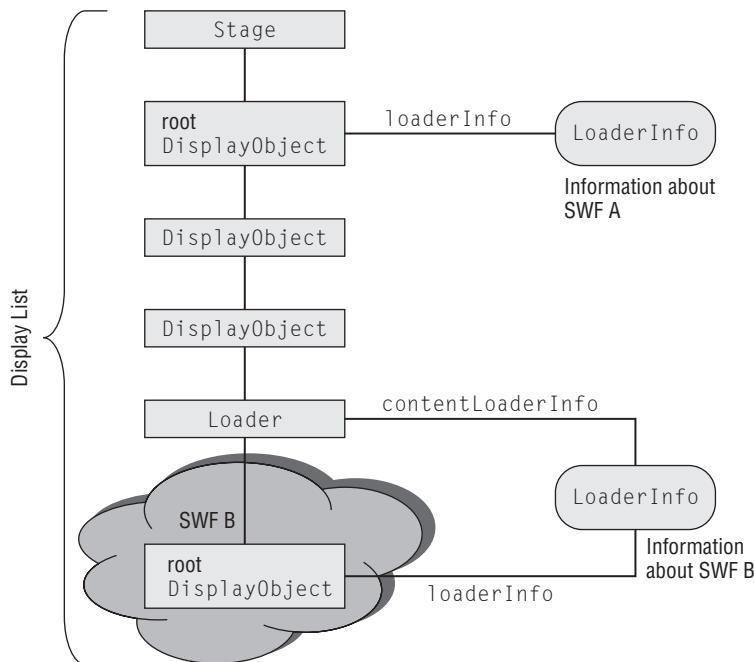
Before I get into the details of `LoaderInfo`, let's clear up some confusing points about `LoaderInfo` objects. These are some of the stickiest issues in the whole Flash Player API, so heed well!

LoaderInfo versus ContentLoaderInfo

`Loader` instances contain two public `LoaderInfo` properties: `loaderInfo` and `contentLoaderInfo`. The relationship of these two `LoaderInfo` instances is shown in Figure 27-1.

FIGURE 27-1

LoaderInfo objects in a display list

The `loaderInfo` property is inherited from `DisplayObject`. That's right — *all* display objects have a `loaderInfo` property, which is a read-only reference to a `LoaderInfo` object. The `loaderInfo` property is set on any object on the display list of a given SWF. The property is defined on `DisplayObject` so that you can easily tell which SWF any display object belongs to.

This isn't the same `LoaderInfo` object that provides information about an image file you load dynamically. The typical application of this `loaderInfo` property is to find out meta-information about the currently running SWF file, often the parameters it has been passed either through FlashVars or in a query string. (See more about passing parameters to SWF files in Chapter 42, "Deploying Flash on the Web.") You can also use it to find out the version of Flash Player the SWF was published for, the original published dimensions and frame rate, and so on.

`Loader` objects, as subclasses of `DisplayObjects`, must also have this `loaderInfo` property. However, this property does *not* contain information relating to the file they're loading. It is the same `loaderInfo` property just described.

For information about the content being loaded, `Loaders` expose a separate property that is also a `LoaderInfo` object: `contentLoaderInfo`. Because it's also an instance of the `LoaderInfo` object, you can get all the same kind of information about images being loaded and any running SWF. This distinction isn't always obvious, and it's easy to forget about, so it's a common stumbling block. Just remember that for a `Loader`, you are always interested in its *content*. You'll see the term "content" applied to mean "that which a `Loader` loads" throughout the `Loader` API.

Loading Events

Another common problem is that of subscribing to load events. Naturally, you'll want to be notified by a `Loader` when it is done loading an image, when the loading fails to complete, and so on. You might do this by subscribing to events on the `Loader`. It does inherit from `EventDispatcher`, after all. But all the interesting events you want to listen for are actually dispatched by the associated `LoaderInfo` object, not the `Loader` itself!

This can become a bit of a nasty slip-up as well, especially because there is no compile-time or run-time check for event subscriptions. It's perfectly legal to subscribe to events that will never be broadcast. So the outcome of subscribing to load events on the `Loader` instance isn't an error, but merely a confusing absence of events.

Remember to subscribe to load events broadcast by the `contentLoaderInfo` property of a `Loader`, not by the `Loader` itself. Now that you know how to use `contentLoaderInfo`, you can look at some of the various properties and events of a `LoaderInfo` object to see what kind of loading information you can use.

The LoaderInfo Class

First I'll cover some of the interesting properties of `LoaderInfo`. Because this class provides information about a SWF or image that it can't change, all the properties are read-only. These properties can refer to either an image or a SWF that's being loaded, or a currently running SWF.

- `width, height` — Store, as `ints`, the dimensions of the image or published size of the SWF.
- `bytesLoaded, bytesTotal` — Return the number of bytes in the file and how many have been thus far loaded. Note that too early in the load process, before Flash Player has received a size for the file, `bytesTotal` can return 0 even though the file is actually large, so be careful when using this for calculations.
- `contentType` — The MIME type of the image file being loaded. May be `null` before enough is loaded to determine.
- `frameRate` — For a SWF, the frame rate it was published at.
- `parameters` — For a SWF, stores, as an `Object` with named properties, the FlashVars and URL parameters passed in by the browser.

There are some properties that are useful specifically when loading a SWF from the internet. I'll cover these in the section "Loading external SWFs."

These events allow you to track the progress of the `Loader`'s load process:

- `Event.COMPLETE` — Signifies that the load completed successfully.
- `IOErrorEvent.IO_ERROR` — Signifies that some networking error occurred. The load operation will fail.
- `SecurityErrorEvent.SECURITY_ERROR` — Signifies that the file could not be loaded because it would violate the security policy. The load operation will fail. See "Understanding Flash Player Security" later in this chapter.
- `Event.OPEN` — Dispatched when the load begins.
- `HTTPStatusEvent.HTTP_STATUS` — Dispatched when Flash Player receives an HTTP response. The `status` property of the event object contains the HTTP status code. In case of an error code, sometimes an `IOErrorEvent` is broadcast instead, so don't rely on this event to report HTTP error codes accurately.
- `ProgressEvent.PROGRESS` — Dispatches every time more data is received from an active load. The `bytesLoaded` and `bytesTotal` properties of the event object tell you exactly how much progress has actually occurred.
- `Event.UNLOAD` — Dispatched when the loaded content is removed by using the loader to load another image, or with the `Loader` methods `unload()` or `unloadAndStop()`.

Caution

Error events, if not handled, can interrupt normal processing of ActionScript and might show an error dialog box in debug versions of Flash Player. Check Chapter 24, "Errors and Exceptions," for more information. Because of this, it's good practice to always handle the `LoaderInfo`'s error events. ∎

Example 27-2 puts together some of what you've learned about `Loader` and `LoaderInfo` to track the load progress of a large image.

EXAMPLE 27-2   `http://actionscriptbible.com/ch27/ex2`

Tracking Load Progress

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;
  import flash.net.URLRequest;

  public class ch27ex2 extends Sprite {
    protected var preloadTF:TextField;
    protected var l:Loader;
    public function ch27ex2() {
      l = new Loader();
```

continued

EXAMPLE 27-2 *(continued)*

```
    //photo   (CC-BY) Roger Braunstein
    //source  http://www.flickr.com/photos/rogerimp/2940373537/
    var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
    l.load(new URLRequest(url));

    //this text field will show the progress of the loading process.
    preloadTF = new TextField();
    preloadTF.defaultTextFormat = new TextFormat("_sans", 12, 0);
    preloadTF.autoSize = TextFieldAutoSize.LEFT;
    addChild(preloadTF);

    l.contentLoaderInfo.addEventListener(Event.COMPLETE, onComplete);
    l.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS, onProgress);
    l.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR, onError);
  }

  protected function onComplete(event:Event):void {
    var info:LoaderInfo = LoaderInfo(event.target);
    preloadTF.text = "loaded " + info.width + "x" + info.height + " " +
      (info.bytesTotal>>10) + "kb" + " image of type " + info.contentType;
    l.y = preloadTF.textHeight + 4;
    addChild(l);
  }

  protected function onProgress(event:ProgressEvent):void {
    var percent:Number = event.bytesLoaded / event.bytesTotal * 100;
    preloadTF.text = percent.toFixed() + "%";
  }

  protected function onError(event:ErrorEvent):void {
    preloadTF.text = event.text;
  }
 }
}
```

The example demonstrates some of the most common tasks regarding loading information. You know when the image is done loading because you subscribed to the `Event.COMPLETE` event. You can notify the user about how the load is progressing because you subscribed to the `Progress Event.PROGRESS` event. You know various information about the loaded file from its `LoaderInfo` object, and you handle errors by catching `IOErrorEvent.IO_ERROR` events. The example works correctly because it uses the `Loader`'s `contentLoaderInfo` property for loading-related events and properties.

Just like you can get a reference to the object that dispatched an event by querying the `target` property of an event object, you can work backward from a loading event to the `Loader`. In the example,

the `Loader` is an instance variable, but if it were not, you could get back to it by using the `loader` property of a `LoaderInfo` object:

```
var loader:Loader = LoaderInfo(event.target).loader;
```

Getting Loaded Content

The `Loader` object, as you've seen, can draw itself since it is a `DisplayObject`. You can add it to the display list before you even ask it to load an external image with `load()`. It has no visual content or dimensions of its own. When an image or SWF is loaded, it is added to the `Loader` as a child display object, just like creating an empty `Sprite` and adding a visible display object to its list. In many cases, it's perfectly acceptable to consider the `Loader` the content. The intermediary, otherwise empty `Loader`, like a container `Sprite`, has no impact on the appearance or behavior of its contents. You can move, scale, rotate, and filter the `Loader` because these transforms affect not just the `Loader` but its children.

Should you need access directly to the loaded image, however, `Loader` makes this possible. Simply use the `content` property of a `Loader` instance. This property is read-only, and it's typed as a `DisplayObject`. The actual type of the content depends on what you loaded in (although it will always be a subclass of `DisplayObject`, of course). `Loader` will do the correct thing for the file type you load. Should you load a bitmap image, it will be a `Bitmap`. If you load a SWF, `content` will be either the type of the root display object, or an `AVM1Movie` if you load a SWF published using ActionScript 1.0 or ActionScript 2.0. You can't count on the `content` property being defined until the load completes successfully.

By accessing their `content` property, you can use `Loader` instances to provide your application with graphical assets. For example, you can get the raw bitmap data out of an image file by loading it and then using `content` to get down to a `BitmapData` instance. Example 27-3 uses this technique to draw a swatch with the color the mouse is over.

EXAMPLE 27-3   `http://actionscriptbible.com/ch27/ex3`

Getting at a Loader's Content

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  import flash.text.*;

  public class ch27ex3 extends Sprite {
    protected var bmpData:BitmapData;
    protected var swatch:Swatch;

    public function ch27ex3() {
      var l:Loader = new Loader();
      //photo   (CC-BY) Roger Braunstein
      //source  http://www.flickr.com/photos/rogerimp/188712483/
      var url:String = "http://actionscriptbible.com/files/pastels.jpg";
      l.load(new URLRequest(url), new LoaderContext(true));
```

continued

EXAMPLE 27-3 *(continued)*

```
      l.contentLoaderInfo.addEventListener(Event.COMPLETE, onComplete);
      l.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR, onError);
      addChild(l);
    }
    protected function onComplete(event:Event):void {
      var loader:Loader = LoaderInfo(event.target).loader;
      if (loader.content is Bitmap) {
        bmpData = Bitmap(loader.content).bitmapData;
        swatch = new Swatch();
        addChild(swatch);
        addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
      }
    }
    protected function onError(event:ErrorEvent):void {
      trace(event.text);
    }

    protected function onMouseMove(event:MouseEvent):void {
      var mx:int = stage.mouseX, my:int = stage.mouseY;
      if (mx >= 0 && mx < bmpData.width && my >= 0 && my < bmpData.height) {
        swatch.x = mx;
        swatch.y = my;
        swatch.color = bmpData.getPixel(mx, my);
        event.updateAfterEvent();
      }
    }
  }
}

import flash.display.Shape;
class Swatch extends Shape {
  public function set color(c:uint):void {
    graphics.beginFill(c);
    graphics.lineStyle(0, 0xffffff, 1, true);
    graphics.drawRect(10, 10, 25, 15);
    graphics.endFill();
  }
}
```

You'll learn more about BitmapData and the graphics property used to draw the swatch in Part VIII of this book. The important aspect of this example is that the Loader not only loads an image and adds it to the display list, but it provides access to the display object that Loader constructs from the file. Not only is the Loader displaying an image of brightly colored pastels, it's done so totally within the normal Flash Player API — the loaded image is a Bitmap in the display list of the Loader.

You might notice a second argument passed to `load()` that I didn't cover. This was required for the example to be able to access the data from the loaded image; you'll learn more about why in the section "Understanding Flash Player Security" later in this chapter.

You can, like any other display object, remove the loaded content from its `Loader` and place it elsewhere in another display list. The `Loader` won't lose track of it; it will still be the `content` property of the `Loader`, and it will still disappear if you `unload()` the `Loader`.

Tip

When unloading sounds and movies loaded using a `Loader`, you typically have to be very careful to remove all event listeners and `null` all references to the media to be a hundred percent certain that the media stops playing. Otherwise, you may notice that the media continues playing in the background despite your efforts.

In Flash Player 10.1 and later, you can use the hammer of doom on your loaded media with `Loader`'s `unloadAndStop()` method, which assures your media unloads properly despite normal garbage collection rules. ■

Loading External SWFs

Again, if all you want to do with a loaded SWF is put it on the stage, you can do it as succinctly as in Example 27-1; loading works the same no matter what image format you load.

However, the ability to load SWFs at runtime provides some interesting opportunities to break out applications into multiple files or interface between several separate programs.

Instantiating Classes from External SWFs

Both code, and assets like graphics, sounds, videos, and fonts, are represented by classes in Action-Script 3.0. Recall from Chapter 16 that assets published in a SWF that are exported for ActionScript become classes that extend the appropriate type of asset, whether you publish them from Flash Professional or use Embed metadata tags with the Flex SDK compiler.

Typically, you create a new instance of a class with the `new` operator, as in `new Sprite()`. But you can also use `new` with a variable of type `Class`, which stores a class reference. You can get a class reference by assigning it directly to a known class:

```
var rectClass:Class = Rectangle;
var rect:Rectangle = new rectClass();
trace(rect); //(x=0, y=0, w=0, h=0)
```

You can also look up classes by their name, using the `getDefinitionByName()` function in the `flash.utils` package.

```
var pointClass:Class = Class(getDefinitionByName("flash.geom.Point"));
var point:Point = new pointClass();
trace(point); //(x=0, y=0)
```

You have to cast the definition to a `Class` because you can also use the `getDefinitionByName()` function to look up other public items like functions, values, and namespaces.

When you have multiple SWFs loaded, the system must figure out how to partition the definitions in the SWFs. Class uniqueness is only guaranteed for one SWF. The compiler can't, and shouldn't, try to ensure that no two SWFs ever have classes with the same name. So when you load in a new SWF,

by default, it gets a unique domain to define namespaces and classes within. This domain is called an `ApplicationDomain`, and you can access and manipulate it in ActionScript. In fact, when you tell a `Loader` to `load()` a SWF, you can actually specify what `ApplicationDomain` Flash Player should place the definitions from that SWF into, including the option of merging the classes into the current domain, in which case definitions from the newly loaded SWF may override existing definitions.

For instance, if you declare `PILL` to be blue in your SWF and load in a SWF that defines `PILL` as red, these `PILL`s can either exist in separate domains, alternate universes in which the `PILL` is to everyone's knowledge definitely blue or definitely red, depending on which domain you are running in; or the newly loaded one can take precedence, merging the two universes and overwriting values of `PILL` with the newly loaded red-pill reality.

When you use `getDefinitionByName()`, it's looking for definitions in the current `ApplicationDomain`. What determines the current domain? Simply where the currently executing code originated — which SWF it was compiled into. You can access the current domain, by the way, with the static property `ApplicationDomain.currentDomain`.

Things get interesting when you access other SWFs' application domains. The `LoaderInfo` object associated with a SWF contains a reference to the SWF's `ApplicationDomain`. And, using methods of the `ApplicationDomain`, you can query and retrieve class references from the SWF. The result is that you can load in a SWF that is not intended to be added to the stage, but rather acts as a library of code or assets. The code and assets can be retrieved dynamically by any code at runtime, provided the code can find the loaded SWF's application domain.

In Example 27-4, two library SWFs were prepared, one using the Library in Flash Professional, and one using the `Embed` metadata tag in Flash Builder. In both SWFs, an icon was linked to the class `assets.Icon`. The example loads the two SWFs dynamically, retrieves references to classes called `assets.Icon` from both application domains, and attaches them to the stage. This demonstrates that the same class name can resolve to different classes in different domains; although the SWFs both define a class named `assets.Icon`, each presents a different graphic. Furthermore, `describeType()` is used to get runtime information about each class. If you parse out the inheritance diagram from each SWF's version of `Icon`, you see that assets embedded with Flash Professional and with Flash Builder have different structures.

EXAMPLE 27-4   `http://actionscriptbible.com/ch27/ex4`

Retrieving Data from Multiple Domains

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.net.URLRequest;
  import flash.system.*;
  import flash.text.*;
  import flash.utils.describeType;

  public class ch27ex4 extends Sprite {
    public function ch27ex4() {
```

```
    var l:Loader;
    l = new Loader();
    l.load(new URLRequest("http://actionscriptbible.com/files/icon-fl.swf"),
      new LoaderContext(true, null, SecurityDomain.currentDomain));
    l.contentLoaderInfo.addEventListener(Event.COMPLETE, onComplete);

    l = new Loader();
    l.load(new URLRequest("http://actionscriptbible.com/files/icon-fx.swf"),
      new LoaderContext(true, null, SecurityDomain.currentDomain));
    l.contentLoaderInfo.addEventListener(Event.COMPLETE, onComplete);
  }

  protected function onComplete(event:Event):void {
    var loaderInfo:LoaderInfo = LoaderInfo(event.target);
    var iconClass:Class =
      Class(loaderInfo.applicationDomain.getDefinition("assets.Icon"));
    var icon:DisplayObject = DisplayObject(new iconClass());
    addChild(icon);

    var tf:TextField = new TextField();
    tf.defaultTextFormat = new TextFormat("_sans", 11, 0);
    tf.width = 150; tf.height = 400; tf.y = 130;
    tf.multiline = tf.wordWrap = true;
    addChild(tf);

    tf.text = loaderInfo.url + "\n\n" + typeInfo(icon);
    if (numChildren > 2) tf.x = icon.x = 160;
  }

  protected function typeInfo(obj:Object):String {
    var info:XML = describeType(obj);
    var ret:String = info.@name;
    for each (var extendsNode:XML in info.extendsClass) {
      ret += " \u2192 " + extendsNode.@type.toString().replace(/^.*::/, "");
    }
    return ret;
  }
  }
}
```

Rather than using the package-scoped function `getDefinitionByName()`, Example 27-4 uses the method `getDefinition()` of the correct `ApplicationDomain` instance to retrieve a class reference from a specific domain. If you had used the package-level function, Flash Player would look for an `assets.Icon` class in the current application domain and fail to find one.

Because Flash Professional makes it easy to visually manage and tweak the compression settings of bitmap and vector graphics, sounds, videos, and fonts with its Library panel, I use Flash Professional to compile SWFs containing all the assets I'll need for an application. This way, I can load the site's

assets at runtime with a `Loader` and query all of them from one unified `ApplicationDomain`. The relatively expensive operation of recompressing and recompiling the assets into a new SWF only needs to be done when I update assets, and the application SWF can be code-only so that it compiles quickly and is lightweight.

Tip

If you want to share a project between developers on multiple computers, especially multiple platforms where font handling differs slightly, you might consider creating one SWF that contains just the fonts. By isolating this SWF from all other assets and code, you can ensure that people working on the code and people working on the other assets don't need to have the fonts correctly installed. ■

Interacting with Loaded SWFs

Because the content of a loaded image or SWF is placed inside the `Loader`, you can interact with a loaded SWF through the display list. You can also instantiate classes from its application domain as just shown. In either scenario, however, your code may not know about the classes in the loaded SWF; the whole point is that they are separate, after all.

Note that in Example 27-4, you never created a variable typed as `Icon`, even though that's the class you created, because `Icon` is a class that was only defined in the loaded SWFs. Instead, you typed the `Icon` instance as a `DisplayObject` and forced it to be such with a cast. You can do this when you know for certain that `Icon` should extend `DisplayObject`, and you need it to be a `DisplayObject` to add it to the display list. This usually works fine for asset classes whose supertypes are known ahead of time.

But what if you need to call methods of a class from another domain, whose definition you don't have in your own codebase? If you load a SWF `zoo.swf` with a class `Hare` in it, although you can get the definition of a `Hare`, a class reference to a `Hare`, and even a `Hare` itself from the loaded `zoo.swf`, the code you write to interact with the `Hare` won't actually have a `Hare` class available. This is actually the perfect place to use untyped variables. When you use an untyped variable, no typechecking is done at compile time. You can call a method or access a property on an untyped variable with any name. Whether that method or property actually exists is a question left until runtime. So you can call `hop()` on what you believe to be a `Hare` object, even if you don't have a definition of `Hare` to code against.

```
var hareClass:Class = Class(zooApplicationDomain.getDefinition("Hare"));
var hare:* = new hareClass();
hare.hop();
trace(hare.whiskerLength);
```

If you use untyped variables, your code can easily interface with classes from dynamically loaded SWFs, while skipping type checks from the compiler. Of course, you should be doubly careful to do runtime error handling in these cases.

Calling code in other SWFs is subject to the Flash Player security model, which is covered later in this chapter.

Loading AVM1 SWFs

If you load a SWF that's published using ActionScript 1.0 or ActionScript 2.0 — for Flash Player versions less than 9 — this SWF runs in a completely different virtual machine in Flash Player, the legacy AVM1 (ActionScript Virtual Machine version 1). All ActionScript 3.0 programs run in AVM2. Because

it's running in a completely different environment, you won't be able to interface with the code in an AVM1 SWF. You won't be able to dive into its display list, instantiate classes from it, or call methods on its root object. However, it will still run and display as intended. To the Flash Player API in ActionScript 3.0, the loaded movie will appear as a black box, an `AVM1Movie` instance, which is a subclass of `DisplayObject`.

There are some ways to find out information about a loaded SWF, besides the fact that after a `Loader` loads in an AVM1 movie, its `content` property is an `AVM1Movie`. These properties of a `LoaderInfo` object relate to its version:

- `swfVersion` — The major version of Flash Player the SWF was published for, as a `uint`. These values are correlated to static constants defined by `SWFVersion` such as `SWFVersion.FLASH10`, which equals 10.

- `actionScriptVersion` — The language used when the SWF was compiled, as a `uint`. The values are correlated to static constants of `ActionScriptVersion`, such as `ActionScriptVersion.ACTIONSCRIPT3` (which has the value 3). The `actionScriptVersion` property only distinguishes between older versions of ActionScript (which are all lumped into "ActionScript 2.0") and ActionScript 3.0. Essentially, it tells you whether the SWF is running in AVM1 or AVM2.

Using these properties at runtime, you can determine if the loaded SWF is compatible and choose what to do with it.

There is one way that communication can be established between an AVM1 and an AVM2 SWF, and that is through a `LocalConnection`. Because this connection is not code that is communicating directly, but one side broadcasting and another side receiving over an opened channel, you can use this to communicate with legacy code if you absolutely must. `LocalConnection` is covered in Chapter 44, "Local Connections Between Flash Applications."

Using Loader to Interpret Files in Memory

As hinted at, you can bypass the loading facilities of `Loader` and use its file decoding abilities directly. This is great if you

- Load the contents of an image file stored on the local system.

- Decode data from another format (zipped data, Base-64 encoded).

- Extract a binary image file from a larger data stream.

- Manufacture a file by "hand" in memory (for a time, dynamic sound was achieved by using code to craft legal SWF files with the correct sound embedded, and then loading and playing these SWFs as they are generated).

If you can get the binary contents of an image file into a `ByteArray` using these approaches or one of your own design, you can display it using a `Loader` instance by passing it to `loadBytes()`.

As you've seen, `Loader` is a versatile, multipurpose class. It loads data from the network or memory, decodes it, and displays it on-screen.

URLLoader

Although `Loader` is great for loading images, use the `URLLoader` class to load any kind of data over HTTP. `URLLoader` doesn't decode or display the contents of an HTTP response; instead, it provides them to you raw. So you can use `URLLoader` to write custom communications that occur over HTTP, from something as simple as loading an XML file to something as complex as implementing a web service, which I'll start to get into in the next chapter.

Loading a File

The `URLLoader` performs an HTTP request, which you specify with a `URLRequest` object, and receives an HTTP response. The `URLLoader` is an `EventDispatcher`, so it uses events to notify any listeners when it begins loading data, if there's an error, what the status of the HTTP request is, when there's any progress, and finally, when it's complete. These events are identical to those dispatched by a `Loader`'s `LoaderInfo` object, although in this case, there is no intermediary object. The events are dispatched directly by the `URLLoader`.

You've already used a `URLLoader` to get data for previous examples. Recall Example 17-13 (`http://actionscriptbible.com/ch17/ex13`). In this example, you loaded a chapter of *Alice's Adventures in Wonderland* by using the following code:

```
var loader:URLLoader = new URLLoader(new URLRequest(
  "http://actionscriptbible.com/files/alice-ch1.txt"));
```

The `URLLoader` constructor can also be called with no parameters, in which case you would pass the `URLRequest` to its `load()` method when you are ready to start loading. When passed directly to the constructor, loading begins automatically.

After you begin loading (really after you begin sending the HTTP request), you're going to need to know when loading is complete (when you can access the full contents of the HTTP response) so that you can do something interesting with what you've loaded.

In the example, you subscribed to the `Event.COMPLETE` event:

```
loader.addEventListener(Event.COMPLETE, onLoadComplete);
```

After a `URLLoader` has finished loading, you can get at what it loaded through the `data` property. This is just the contents of the HTTP response; all the headers and meta-information are handled by `URLLoader`, giving you direct access to the important part: the data. In this case, this is just plaintext. You retrieve it like so:

```
protected function onLoadComplete(event:Event):void {
  tf.text = URLLoader(event.target).data;
}
```

Simple! You just assign the `data` property, which is untyped but contains a `String` value, to the `text` property of the text field `tf`, which is a `String`.

Loading and Canceling

You can start a `URLLoader` loading by calling its `load()` method, passing the `URLRequest` describing the HTTP request. Stop a load operation with the `close()` method.

Tracking Load Progress

The `URLLoader` class dispatches most of the same events as a `LoaderInfo` class. You can review these in the earlier section "The LoaderInfo class." The supported events are

- `Event.COMPLETE` — The load completed.
- `HTTPStatusEvent.HTTP_STATUS` — The HTTP status code is received.

- `IOErrorEvent.IO_ERROR` — A communication error occurred.
- `Event.OPEN` — A connection has been opened.
- `ProgressEvent.PROGRESS` — More data has been loaded.
- `SecurityErrorEvent.SECURITY_ERROR` — The load operation violated Flash Player's security policy.

Additionally, the `URLLoader` object exposes the `bytesLoaded` and `bytesTotal` properties (also available in a `ProgressEvent` event object).

Loading Different Formats

You can use a `URLLoader` to load *any* kind of data. The API distinguishes among the types of data in three broad categories: text, binary, or URL-encoded variables. Now, to be sure, URL-encoded variables are also a kind of text-based data, but Flash Player supports them specially. To get the data from a completed load operation, as you've seen, you simply access the `URLLoader`'s `data` property. Before you start loading, however, you should provide a hint to the `URLLoader` as to what general category of data it's going to be receiving so that the data is interpreted properly and the `data` property is populated with the correct type. You do this by setting the `URLLoader`'s `dataFormat` property. `dataFormat` is a string that can have one of these three values, as defined by the enumeration `URLLoaderDataFormat`:

- `URLLoaderDataFormat.TEXT` — Interpret the response as text. The `data` property will be of type `String`. This is the default.
- `URLLoaderDataFormat.BINARY` — Interpret the response as binary data. The `data` property will be of type `ByteArray`.
- `URLLoaderDataFormat.VARIABLES` — Interpret the response as URL-encoded variables. The `data` property will be of type `URLVariables`.

Once you've chosen the proper category for the data, you may still need to parse it. Let's look at how to load some common types of data.

Loading XML

XML is a text-based format, so load the file as text, and then use the top-level `XML()` cast-like function to convert the `String` into `XML`, as shown in Example 27-5.

EXAMPLE 27-5 `http://actionscriptbible.com/ch27/ex5`

Loading and Reading XML

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  import flash.net.*;

  public class ch27ex5 extends Example {
    public function ch27ex5() {
      var URL:String = "http://actionscriptbible.com/files/hand.xml";
      var loader:URLLoader = new URLLoader(new URLRequest(URL));
```

continued

(continued)

```
    loader.dataFormat = URLLoaderDataFormat.TEXT;
    loader.addEventListener(Event.COMPLETE, onLoadComplete);
  }
  protected function onLoadComplete(event:Event):void {
    trace("Done loading XML.");
    var loader:URLLoader = URLLoader(event.target);
    var hand:XML = XML(loader.data);
    var card:XML = hand.card[0];
    trace("Your first card is a " + card.@type + " of " + card.@suit);
    trace(hand.toXMLString());
  }
 }
}
```

Loading CSS

CSS is another text-based format. Example 27-6 uses the `StyleSheet` class and its `parseCSS()` method.

EXAMPLE 27-6 `http://actionscriptbible.com/ch27/ex6`

Loading and Parsing CSS

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  import flash.net.*;
  import flash.text.StyleSheet;

  public class ch27ex6 extends Example {
    public function ch27ex6() {
      var URL:String = "http://actionscriptbible.com/files/example.css";
      var loader:URLLoader = new URLLoader(new URLRequest(URL));
      loader.dataFormat = URLLoaderDataFormat.TEXT;
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onLoadComplete(event:Event):void {
      trace("Done loading CSS.");
      var loader:URLLoader = URLLoader(event.target);
      var css:StyleSheet = new StyleSheet();
      css.parseCSS(loader.data);
      trace("Paragraphs use color " + css.getStyle("p").color);
      trace(loader.data);
    }
  }
}
```

Loading a Binary File

It's not realistic to include the full source of an example that loads and parses a binary file format (for which Flash Player doesn't already have a decoder). Typically, these rely on libraries you write yourself or link in. For instance, you can get libraries to decode ZIP archives (FZip, `http://codeazur.com.br/lab/fzip/`), crack open SWF files (swfassist, `http://www.libspark.org/wiki/yossy/swfassist`), parse VCards (Adobe corelib, `http://code.google.com/p/as3corelib/`), and the list goes on. But unless the library does its own loading, the process for loading binary-format files is identical: load the file as a binary type, and pass along the `URLLoader`'s `data` property as a `ByteArray` to whatever library you will use to decode the file.

In Example 27-7, instead, you'll load a JPEG file and look for a part of the JPEG header called the JFIF (for JPEG File Interchange Format) marker. This marker contains some essential information about the image and would be used by a JPEG parser, if you were for some reason inspired to write one.

EXAMPLE 27-7 `http://actionscriptbible.com/ch27/ex7`

Loading and Reading from a Binary File

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  import flash.net.*;
  import flash.utils.ByteArray;

  public class ch27ex7 extends Example {
    public function ch27ex7() {
      var URL:String = "http://actionscriptbible.com/files/caviar.jpg";
      var loader:URLLoader = new URLLoader(new URLRequest(URL));
      loader.dataFormat = URLLoaderDataFormat.BINARY;
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onLoadComplete(event:Event):void {
      trace("Done loading JPEG.");
      var loader:URLLoader = URLLoader(event.target);
      var data:ByteArray = ByteArray(loader.data);
      data.position = 0x06; //the JFIF id string starts 6 bytes into the file
      //the JFIF ID is the zero-terminated string "JFIF", so if we try to
      //read in a string starting here, we should find "JFIF".
      var markerString:String = data.readUTFBytes(4);
      trace("Marker: " + markerString);
    }
  }
}
```

Loading URL-Encoded Variables

In actuality, URL-encoded variables are even more useful when sending data than receiving data, because they are the standard way to communicate arguments to a server-side script. Both the GET

and POST HTTP methods allow you to send arguments in the HTTP request. The `URLVariables` class simplifies this process.

```
make=Panasonic&model=PT%2DAE2000U&resolution=1080p
```

These name-value pairs are an example of URL-encoded variables — simply a string that encodes variables with `name=key` pairs separated by ampersands (&). The names and values in the string are escaped using URL encoding to ensure that the full string can be used in a URL and to ensure that reserved characters like & and = are not used within the string, which would mess up the encoding. Here a hyphen (-) was replaced by its URL-encoded equivalent, `%2D`. There is plenty of further reading on URL encoding available on the internet. Thankfully, you don't have to worry too much about the encoding because you have `URLVariables` to take care of it for you.

Tip
Two functions in the default package help with URL encoding: `escape()` and `unescape()`. These convert strings to and from their URL-encoded versions. ∎

To use `URLVariables`, simply set and retrieve properties on the `URLVariables` instance by name. It works just like an `Object` in that it's declared as dynamic so you can access properties of a `URLVariables` by any old name. Let's re-create the preceding query string:

```
var vars:URLVariables = new URLVariables();
vars.make = "Panasonic";
vars.model = "PT-AE2000U";
vars.resolution = "1080p";
trace(vars.toString()); //make=Panasonic&model=PT%2DAE2000U&resolution=1080p
```

If you're going to receive data in a URL-encoded string, you can set the `dataFormat` property of your `URLLoader` to `URLLoaderDataFormat.VARIABLES`, as Example 27-8 demonstrates.

EXAMPLE 27-8 `http://actionscriptbible.com/ch27/ex8`

Loading URL-Encoded Variables

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  import flash.net.*;

  public class ch27ex8 extends Example {
    public function ch27ex8() {
      var URL:String = "http://actionscriptbible.com/files/projector.txt";
      var loader:URLLoader = new URLLoader(new URLRequest(URL));
      loader.dataFormat = URLLoaderDataFormat.VARIABLES;
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
```

```
    }
    protected function onLoadComplete(event:Event):void {
      trace("Done loading variables.");

      var loader:URLLoader = URLLoader(event.target);
      var vars:URLVariables = URLVariables(loader.data);
      trace("It's a " + vars.make + " " + vars.model);
      trace(vars);
    }
  }
}
```

Sending Variables with URLRequest

Now, in addition to loading information from external sources, you can send information to external sources that will allow you to receive data based on what you sent. I'll go much more deeply into this in the next chapter, but let's look at the basics here.

GET and POST are two ways of sending information between web pages via the browser. The main difference between these two HTTP methods is that GET puts the variables in the URL, whereas POST adds them to the body of the HTTP request. Let's look at a GET request first:

```
http://actionscriptbible.com/do?user=me&login=true
```

What this is telling you is to not only access this page, but also to take along with you two GET variables: one called `user`, which is set to `me`, and one called `login`, which is set to `true`. The actual HTTP request for this looks like this (simplified):

```
GET /do?user=me&login=true HTTP/1.1
Host: actionscriptbible.com
User-Agent: Mozilla/4.0
```

The one thing to always remember about GET is that it is *not* safe from prying eyes. Anyone can see the information sent in a GET in your URL, which hopefully would not be a credit card number or other valuable information.

A POST doesn't show up in the URL; it is hidden in the actual HTTP request. Caution, however! Although variables are not transparent in a POST, this doesn't make them any more secure. A simplified POST request looks like this:

```
POST /do HTTP/1.1
Host: actionscriptbible.com
User-Agent: Mozilla/4.0
Content-Length: 19
Content-Type: application/x-www-form-urlencoded

user=me&login=true
```

As you can see, the information and its encoding are the same, but the method that you're sending it in is not. You set the method that you want to use in the method property of the URLRequest. The possible values — GET and POST — are available as constants defined by URLRequestMethod.

```
var request:URLRequest = new URLRequest("http://actionscriptbible.com/do");
request.method = URLRequestMethod.GET;
```

Now that you've established that you're using the GET HTTP method, you can set some data to go over with this request using the data property. The property accepts binary data as a ByteArray, URL-encoded variables as a String or URLVariables instance, or any other text as a String.

```
request.data = "name=me&login=true";
```

Request and Response with URLLoader

Now you can use the URLLoader class just like always to send the request (now carrying a payload of data, and using the method you specified) and load the response, as shown in Example 27-9.

EXAMPLE 27-9 http://actionscriptbible.com/ch27/ex9

HTTP Request and Response

```
package {
  import com.actionscriptbible.Example;
  import flash.events.*;
  import flash.net.*;

  public class ch27ex9 extends Example {
    public function ch27ex9() {
      var url:String = "http://actionscriptbible.com/files/login.php";
      var request:URLRequest = new URLRequest(url);
      request.method = URLRequestMethod.POST;
      var vars:URLVariables = new URLVariables();
      vars.login = true;
      vars.user = "Roger";
      request.data = vars;

      var loader:URLLoader = new URLLoader(request);
      loader.addEventListener(HTTPStatusEvent.HTTP_STATUS, onHttpStatus);
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onHttpStatus(event:HTTPStatusEvent):void {
      trace("HTTP response code " + event.status);
    }
    protected function onLoadComplete(event:Event):void {
      trace("Response follows >>>>>>");
      var loader:URLLoader = URLLoader(event.target);
      trace(loader.data);
    }
  }
}
```

None of the `URLLoader` code is different from any other loading example you've seen up until this point. Where Example 27-9 differs is that the `URLRequest` is jammed full of data, and the POST method is chosen. If you try removing the login variable or changing the name, you should see changes in what the script returns.

When loading a file, recall that you're still making an HTTP request. It's just a GET for the URL you requested. So the technique differs very little; in both cases you make a request and get a reply. It's just that now, communicating with a script instead of asking for a static file, the contents of the response depend on the data found in the request.

In reality, you'd want to make sure that you're always listening for potential errors when you're loading data because you never know what might go wrong: a server down, an ISP conking out, Earth destroyed to make way for a highway, or any number of potential disasters. It's best to be able to say, "Something went wrong" rather than just leaving the user to wonder if you know what you're doing. Remember to listen for `IOErrorEvents` as well as `SecurityErrorEvents`.

You can communicate with many kinds of server-side scripts using GET and POST and loading the results, and in the next chapter I'll expand on this to explore communication with well-structured web services. For now, however, you understand the basics of this communication. You know how to send data, and you know how to listen for the response, and those are powerful things to know.

Request Only with sendToUrl()

In some cases you might not care about what the server replies when you make a request. This can be useful for tracking usage of your application, for updating your status in a multiuser environment, and so on.

Although there's no reason you can't make a request and ignore the response by simply sending it through a new `URLLoader` whose events you ignore and whose existence you promptly forget, Flash Player does give you a tool specifically to send a request without expectation of a reply. This is the function `sendToUrl()`, exposed in the `flash.net` package. This function takes one argument, the `URLRequest`, and returns nothing.

```
sendToURL(new URLRequest("http://example.com/counter?increment=true"));
```

Understanding Flash Player Security

Because so many applications live on the internet now, it's essential that internet-enabled software be well behaved. Our notion of well behaved usually centers on the idea of *sandboxing* — your app shouldn't be able to mess with data from another app, like say, your online banking site. Each site operates in its own little sandbox. And because Flash Player doesn't know or care if you're a good guy or a bad guy when it executes your code, it gives nobody the benefit of the doubt and runs all code with a fair measure of skepticism. Generally, this skepticism applies when things are happening between two domains, like when a SWF from `wonderfl.net` tries to access data on `actionscriptbible.com`. Sometimes Flash Player's security model prevents you from doing things, sometimes it asks the user for permission with a dialog box, but most often, you have to prove to it that what you're doing is expected. For example, the owner of `actionscriptbible.com` has to prove to Flash Player that it's okay for code in `wonderfl.net` to poke around.

Flash Player security is pretty in depth and applies to many parts of the API, and its rules can be a little complicated. In this section, I'll touch on the most important parts, but for

great detail, refer to the AS3LR or Adobe's white papers on Flash Player Security, found at `http://adobe.com/go/fp9_0_security` and `http://adobe.com/go/fp10_0_security`.

The first of the security model's restrictions is that Flash Player is limited by your operating system to certain behavior, much like what your operating system does to other scripts or applications running in your browser. You wouldn't want someone else's JavaScript going and running shell scripts on your computer, and neither would you want Flash Player doing that. The good news is, it can't. That part is quite solid and doesn't affect you much in day-to-day development. The part that does affect you in day-to-day development is the security sandbox, which affects where you can load files from.

The security sandbox exists to ensure that you and only you can specify the locations where content will be loaded into the SWF file that you're hosting. When you put a SWF file on your server and a user downloads it, the Flash Player notes the domain where that SWF is coming from and creates a security sandbox defined by that domain. That means if the user goes and downloads SWF files from my web site with the name `http://www.mysite.com`, the Flash Player will define a security sandbox for `http://www.mysite.com` that all content and data for that player must originate from. It's important to note that `http://www.mysite.com` is not the same as `http://mysite.com` or `http://82.39.28.182`, even if `http://82.39.28.182` is the IP address of your web site. This is nice because it greatly decreases the likelihood that someone will get a virus or something somehow malevolent from viewing Flash content on your site, but it can be a nuisance when you're trying to load XML files or pictures from another site. Flash Player raises a security sandbox exception, and fails to load whether the exception is handled or not. The way around this is to define a `crossdomain.xml` file that lists all the IP addresses and domain names that you plan to access content from for your SWF file. This `crossdomain.xml` file should sit at the server root so the Flash Player can access it and needs to be nothing more than a list of acceptable domains. An example `crossdomain.xml` file might look like this:

```
<?xml version="1.0"?>
<cross-domain-policy>
    <allow-access-from domain="*.mysite.com" />
    <allow-access-from domain="www.anotherofmysites.com" />
    <allow-access-from domain="282.39.28.182" />
</cross-domain-policy>
```

Now PNG files, SWF files, and XML from the host can be loaded by SWFs on any of these three sites. You can place the file at any location on the host and load it manually with `Security.loadPolicyFile()`; if you can, place it at the server root (`http://example.com/crossdomain.xml`) so that Flash Player doesn't have to be told where to find it. This is a good idea when you have multiple SWF files with different sandbox requirements.

If you want to know what the current security settings of a SWF file are, you need only to call `Security.sandboxType()` and see the string that is returned. The possible values are

- `Security.REMOTE` — The SWF file is from an internet URL and follows the domain-based sandboxing described here.
- `Security.LOCAL_WITH_FILE` — The SWF file is running locally, but it has not been trusted by the user and was not published designated for network use. The SWF file can read from local data sources but cannot communicate with the internet.
- `Security.LOCAL_WITH_NETWORK` — The SWF is running locally, has not been trusted by the user, but it was published designated for network use. The SWF can communicate with the internet but not locally.

- `Security.LOCAL_TRUSTED` — The SWF file is a running locally and has been trusted by the user, using the Settings Manager. The SWF file can both read local resources and communicate with the internet.

In addition to restricting the URLs from which the Flash Player may access content, the security mechanism might restrict what loaded content can access or load. When you load a SWF file, you can set the context parameter of the `load()` method of the `Loader` object that is used to load the file. This parameter takes a `LoaderContext` object. When you set the `securityDomain` property of this `LoaderContext` object to `Security.currentDomain`, Flash Player trusts the loaded SWF exactly as much as it already trusts the executing SWF. In other words, it applies the security settings of the current SWF to the loaded SWF. If `securityDomain` is not set (this is the default), the loaded SWF is loaded in a fresh sandbox, and its actions must be approved anew.

Additionally, by setting the `applicationContext` parameter of the `LoaderContext` object, you can load the SWF file as imported media. In this way, the file doing the loading can access objects and classes in the application domain of the loaded SWF. The following code snippet shows loading a SWF file and setting that SWF file's `ApplicationDomain` to be the domain of the loading SWF:

```
public function load(lib:String):void
{
    var loader:Loader = new Loader();
    var request:URLRequest = new URLRequest(swfLib);
    var context:LoaderContext = new LoaderContext();
    context.applicationDomain = ApplicationDomain.currentDomain;
    loader.load(request, context);
}
```

Additionally, the `Security.allowDomain()` method allows the loaded SWF to do things within the domain of the loading SWF file. Call the `Security.allowDomain()` method in the constructor method of the main class of the loaded SWF file, and then have the loading SWF file add an event listener to respond to the `init` event dispatched by the `contentLoaderInfo` property of the `Loader` object. When this event is dispatched, the loaded SWF file has called the `Security.allowDomain()` method in the constructor method, and classes in the loaded SWF file are available to the loading SWF file. The loading SWF file can retrieve classes from the loaded SWF file by calling `Loader.contentLoaderInfo.applicationDomain.getDefinition()`. The following example shows how a SWF loaded into another SWF can use `Security.allowDomain()` to allow the domain of the loading SWF:

```
public function ExampleConstructor() {
    Security.allowDomain("http://www.newdomain.com");
}
```

Summary

- The `URLRequest` class is used to store information about a URL you might want to navigate to, send information to, or get information from, or data that you might want to send.
- To redirect the browser, you use `flash.net.navigateToURL()` and pass in a `URLRequest` object that has the URL that you want to redirect to.

559

- You can also set GET and POST variables to be used by the `URLRequest`. These need to be URL-encoded.

- To load data from a URL, you use a `URLLoader` object. `URLLoader`s dispatch events when they are opened, when they make progress in downloading, when there is an error in downloading, and when they are completed.

- To access the information downloaded by the `URLLoader`, you access the `URLLoader`'s `data` property. You can access this only when the downloading is complete. You determine when a `URLLoader` is finished downloading by listening to the `Event.COMPLETE` event.

- If you are downloading text or XML, you want the `URLLoader`'s `dataFormat` to be set to text, which is the default. If you are downloading `URLVariables`, you want the `URLLoader's dataFormat` to be set to `"variables"`. This creates a `URLVariables` object that you can loop through to access all its properties and data.

- If you aren't interested in any response to your `URLRequest`, you can use the `sendToURL()` method, which sends the request but doesn't listen for information from the user.

Communicating with Remote Services

I n systems design, code that runs on the user's computer is called *client-side,* and code that runs on a server on the internet somewhere is called *server-side*. Applications like single-player games can be entirely client-side, and scheduled tasks like indexing files for search can be entirely server-side, but the majority of applications that live on the internet trade off some of the work between the client and the server. These are client-server applications.

The great thing about client-server applications, and one of the strengths of the internet, is that as long as you agree on a specification for client-server communication, you can use any technology on the client side and any technology on the server side. That's how you've seen web sites transition into web services. What at one time was simply a site for sharing photos, presenting your browser with static pages of photos, is now a photo-sharing service, which you can interface with in a mobile phone app, another web site, or a stand-alone application. As a client-side developer, you use the APIs that the service exposes (and some expose the same services in multiple ways to be even more flexible) to take advantage of that service.

ActionScript 3.0 is a client-side technology: it runs in Flash Player on the end-user's computer. And, like a good client-side technology, it makes it easy to speak a variety of services. There are a ton of ways to get clients and servers to communicate, so in this chapter I'll cover some of the most common ones from the perspective of writing the client-side code. If the choice of which client-server protocol to use is yours, you'll have to make this decision based on deeper knowledge of the technologies than I cover here. I also won't show too much server-side code because there are so many options, and you can better learn those technologies elsewhere. Furthermore, this chapter is meant to be an introduction. I aim for breadth, not depth, while covering a variety of forms of client-server communication. The chapters in this part should give you the tools necessary to work with any of these and provide a springboard if you want to dive in deeper.

FEATURED CLASSES

flash.net.URLLoader

flash.net.URLRequest

flash.net.Socket

flash.net.XMLSocket

flash.net.NetConnection

flash.net.Responder

Web Services Using HTTP

A huge proportion of client-server communication is built on HTTP. Whether you're talking REST or SOAP or XML-RPC, or simply an ad-hoc collection of scripts, a lot of client-server communication is made simple — for both the server and the client — by talking in the same way web browsers and web servers talk. It's kind of a no-brainer: technologies that live on the web already talk in HTTP, so why reinvent the wheel?

Remember from Chapter 27, "Networking Basics and Flash Player Security," that HTTP is the protocol of the web: it defines how to make several requests like GET, POST, PUT, and DELETE, and it defines how the server responds to these requests. If you need to communicate with a server, you can simply access a script hosted at a certain URL, send it arguments with GET or POST, and listen to the response; nothing on top of existing HTTP is needed. An ad-hoc service might provide clients access via a smattering of different scripts at various locations that return information in a variety of formats. If you redesign these scripts such that their URLs, and the HTTP methods you send them, obviously communicate the nature of your request, and they return the resources you request in appropriate and consistent formats, you're building a RESTful system. REST is just an architectural style that fits nicely with HTTP. It's not a technology, and it can be done in any language. Both ad-hoc and RESTful services can be built using just HTTP and nothing more.

Some client-server communication technologies that use HTTP — SOAP and XML-RPC, notably — define protocols "on top" of HTTP. This means that, although you still send requests and receive responses through normal HTTP, the format of those requests and replies is prescribed by the SOAP or XML-RPC spec. These specifications are good because as long as you know the endpoint of any service — the URL that responds to your requests — you can send well-formed requests to it.

Take a look at popular APIs out there: Flickr, Google Maps, Facebook, and so on. Every public web service offers up its API through either REST, SOAP, or XML-RPC, some offering multiple options. So, at least for public services, it's a snap to develop clients in ActionScript 3.0. As you learned in Chapter 27, `URLRequest` and `URLLoader` let you craft and send HTTP requests and load their responses. With just these two classes (okay, usually `XML`, too) you can write the client side of a client-server application. I introduced fairly straightforward use of `URLRequest` and `URLLoader` before. For your convenience, let's review the properties you need to communicate over HTTP with web services:

- `URLRequest.method` — Set the HTTP method to `URLRequestMethod.GET` or `URLRequestMethod.POST`. Unfortunately, using the Flash Player API, you are restricted to these two methods, which might hamper your use of certain fully RESTful APIs. Note that if you make a POST with empty content, Flash Player will use a GET request instead; use dummy content to force the POST.

- `URLRequest.contentType` — Sets the MIME Content-Type header sent in an HTTP request. Defaults to `application/x-www-form-urlencoded`, so this works automatically if you're just sending form variables with POST. Otherwise, set this to correspond to the type of data you're sending in the `URLRequest.data` parameter.

- `URLRequest.data` — Sets the content of an HTTP request. It accepts a `String`, a `URLVariables`, or a `ByteArray` for binary data.

- `URLLoader.dataFormat` — Instructs the `URLLoader` to interpret the server's response as text (`URLLoaderDataFormat.TEXT`), binary data (`URLLoaderDataFormat.BINARY`), or URL-encoded variables (`URLLoaderDataFormat.VARIABLES`).

- `URLLoader.data` — The content of the HTTP response from the server.

Let's look at how you'd use `URLRequest` and `URLLoader` to utilize server-side code.

REST

A RESTful service uses nothing on top of HTTP, as Example 28-1 shows. All you need is to use `URLRequest` and `URLLoader` efficaciously — perhaps some E4X to interpret the responses.

`http://actionscriptbible.com/ch28/ex1`

A REST Service

```
package {
  import com.actionscriptbible.Example;
  import flash.events.*;
  import flash.net.*;
  public class ch28ex1 extends Example {
    private const ENDPOINT:String = "http://geocoder.us/service/rest/geocode";
    public function ch28ex1() {
      geocode("2225 Jackson Ave, Long Island City, NY");
    }
    public function geocode(address:String):void {
      trace("Looking up " + address + "...");
      var req:URLRequest = new URLRequest();
      req.url = ENDPOINT;
      req.method = URLRequestMethod.GET;
      req.data = "address=" + address;
      var loader:URLLoader = new URLLoader(req);
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
      loader.addEventListener(IOErrorEvent.IO_ERROR, onError);
    }
    protected function onLoadComplete(event:Event):void {
      var loader:URLLoader = URLLoader(event.target);
      var response:XML = XML(loader.data);
      var geo:Namespace = response.namespace("geo");
      var lat:Number = parseFloat(response..geo::lat.text());
      var long:Number = parseFloat(response..geo::long.text());
      trace("(" + lat + ", " + long + ")");
    }
    protected function onError(event:Event):void {
      trace("Error!");
    }
  }
}
```

SOAP

Next to REST, SOAP is probably the most common style of web service. In SOAP, requests and responses have a well-defined XML format. Web services in this style clearly define their methods, parameters and types, and return types. You can load a WSDL file to have the service describe itself in full. Because of this structure, SOAP web services are predictable and are usually organized well.

You can either write your own code to craft requests and interpret responses, or you can choose to use a library that encapsulates not only the SOAP structure but the networking code as well, abstracting the web service to an object.

The Flex framework has a fleshed-out framework for SOAP web services, in the `mx.rpc.soap` package. (In fact, it also includes one for REST in the `mx.rpc.http` package, although it may not save you much effort.) The Flex SOAP framework can theoretically be used with an ActionScript-only project, but in reality it takes some hacking, and you're better off just creating a simple Flex shell for your ActionScript application if you want to take advantage of the SOAP classes.

Alternatively, Peter Michels maintains an extremely lightweight Flex-free SOAP framework at `http://as3webservice.googlecode.com`. In Example 28-2, you use this library to simplify a weather web service.

EXAMPLE 28-2 `http://actionscriptbible.com/ch28/ex2`

A SOAP Service

```
package {
  import be.wellconsidered.services.Operation;
  import be.wellconsidered.services.WebService;
  import be.wellconsidered.services.events.OperationEvent;
  import com.actionscriptbible.Example;
  public class ch28ex2 extends Example {
    protected var weatherService:WebService;
    public function ch28ex2() {
      weatherService = new WebService(
        "http://www.webservicex.net/WeatherForecast.asmx?wsdl");
      getWeatherByZipCode("11238");
    }
    public function getWeatherByZipCode(zipCode:String):void {
      trace("Looking up weather in " + zipCode + "...");
      var op:Operation = new Operation(weatherService);
      op.addEventListener(OperationEvent.COMPLETE, onResult);
      op.addEventListener(OperationEvent.FAILED, onFault);
      op.GetWeatherByZipCode(zipCode);
    }
    protected function onResult(e:OperationEvent):void {
      var result:Object = e.data;
      trace("Weather for " + result.PlaceName + ", " + result.StateCode);
      for each (var weatherToday:Object in result.Details) {
        if (weatherToday.Day) {
          trace(weatherToday.Day + ":\t" +
                "Low " + weatherToday.MinTemperatureF +
                "\tHigh " + weatherToday.MaxTemperatureF);
        }
      }
    }
    protected function onFault(e:OperationEvent):void {
      trace("Error: " + e.data);
    }
  }
}
```

XML-RPC

XML-RPC is another contender in the web service arena. It is similar to SOAP in many ways. In fact, SOAP evolved from XML-RPC. XML-RPC is minimal; it is much simpler and more readable than SOAP, but it also lets you specify typed parameters and responses. Just like with SOAP, you can craft and interpret the XML-RPC requests and responses yourself using E4X, or you can use a library to abstract away the details of the service. The Flex framework doesn't provide a premade XML-RPC library, but the community has created a few ActionScript 3.0 XML-RPC libraries, including Akeem Philbert's AS3 RPC Lib (`http://as3-rpclib.googlecode.com`).

Socket Services

Lower-level services on the internet communicate over TCP/IP without using HTTP. HTTP is actually one protocol built on top of TCP/IP. However, it is not the only one, and using Flash Player, you can program clients that will connect to any service that uses TCP/IP sockets (including HTTP; interestingly, nothing's stopping you from writing your own HTTP stack if you don't like the way `URLLoader` works).

Sockets are abstractions that simplify networking code. A socket is a connection between two computers that works just like a stream. You connect up the socket, and information flows in two directions. You can shove data down the pipe, and you can listen for data to come up the pipe. When you're all done, you just disconnect the socket. Simple, no? In reality, there's a lot of code that goes into establishing these connections, keeping them alive, buffering the data that you push down the pipe and sending it along. With sockets, you don't have to worry about any of that.

Think of HTTP. You connect to the server, you send an HTTP request down the pipe, you listen for the response to come back up the pipe, and you disconnect. This is a kind of socket connection. You don't use it like a socket because `URLLoader` handles both the network code and the HTTP protocol that sits on top of it. Just a few services that use TCP/IP include

- Remote terminals (Telnet, SSH) — Allow you to use a command line on a remote computer as if you were there locally. You send keystrokes down the pipe, and the computer sends the output back to you rather than to the screen.

- File transfers (FTP, SFTP, SCP) — After a connection is established to the FTP server, you use simple commands to browse the remote filesystem. Huge chunks of data can be sent up and down the pipe to upload and download files.

- Mail (POP3, IMAP, SMTP) — Every time you check your mail (unless you use webmail, that is), you establish a connection to the mail server and send requests asking it to describe your inbox. Usually it sends you the headers; then you may or may not actually request the messages one by one.

- Chat protocols (Jabber, AIM, MSN, Yahoo!) — Chat is usually achieved by keeping a connection open to a central server. When you send your mom a chat message, you send a command to the server with the message. The server then pushes the message down the socket that connects your mom to the server.

You can create a client for any of these services and more using sockets. All it takes is knowledge of the protocol. You have to be able to send the right commands up to the server and know how to respond to data you receive from the server.

You connect to sockets in ActionScript 3.0 with the `Socket` class. Sockets, as you just learned, are nothing more than bidirectional streams that can be connected and disconnected. So `Socket` implements an input stream and an output stream, and it provides `connect()` and `close()` methods.

565

The input and output stream interfaces, `IDataInput` and `IDataOutput`, should be familiar from `ByteArray`, which also implements them (see Chapter 13, "Binary Data and Byte Arrays").

Note

If you want to write a client for some service in ActionScript 3.0, it's best to do a thorough search online to see if there is already a client — whether completed, under development, or abandoned — that you could use. There's no need to reinvent the wheel unless you have a good reason to. Of course, I give you my unconditional approval to write a client just to help you learn sockets! ∎

You should also be aware that connecting to a service can be a security threat. You or whoever controls the service has to specifically allow access from Flash Player clients, on a domain-by-domain basis, with a `crossdomain.xml` file. For more information, see `http://www.adobe.com/go/flex3_progAS3_security`.

If you decide to implement a common service that uses TCP/IP, chances are you will find an exhaustive description of its protocol as a Request For Comment (RFC). Several sites provide access to these public documents; simply search for "RFC" and the name of the protocol, and you're likely to find it.

These are the universal steps to socket programming and how to achieve them with `Socket`:

- Connect to a service — Specify the server's domain name or IP address, and the port, to the `Socket`'s `connect()` method. Listen for the `Socket` to dispatch an `Event.CONNECT` event, confirming the connection is established. At any time, you can check whether the socket is connected by checking the value of the Boolean property `connected`.

- Send data — As long as you're connected to the server, you can send data at will using `IDataOutput` methods. The protocol itself specifies how to communicate discrete messages to the server — for example, a `null` byte or two line endings after each command. As far as the socket is concerned, you're just sending and receiving bytes.

- Receive data — You have to be connected to read data from the stream, but of course, the server must have sent some stuff for you to be able to read any stuff. You can read and react to the data immediately, which is the normal operating procedure, or you can go ahead and leave the data in the buffer until you are ready for it. New data coming down the pipe from the server will accumulate. Like any input stream implementing `IDataInput`, `Socket` exposes a `bytesAvailable` implicit accessor (property) that tells you how much data there is to read. Furthermore, `Socket` dispatches `ProgressEvent.SOCKET_DATA` events as data is received, so you don't always have to be checking.

- Disconnect — When you're done with the session, make sure to terminate the socket connection. This is pretty easy: just call `close()` on the `Socket`. Not closing a socket you're done with is not only lazy but downright irresponsible! Keep in mind that either party can terminate a socket connection; the server may hang up on you at will. Be sure to listen for this with the `Event.CLOSE` event.

And that's all there is to it! The socket programming API is terribly easy, although conforming to the protocol of a given service may not be.

Let's create a partial HTTP client of your own in Example 28-3, by using sockets.

EXAMPLE 28-3 `http://actionscriptbible.com/ch28/ex3`

Using Sockets

```
package {
  import com.actionscriptbible.Example;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.net.Socket;
  public class ch28ex3 extends Example {
    protected var sock:Socket;
    public function ch28ex3():void {
      sock = new Socket("actionscriptbible.com", 80); //http uses port 80
      sock.addEventListener(Event.CONNECT, onConnected);
      sock.addEventListener(Event.CLOSE, onClosed);
      sock.addEventListener(ProgressEvent.SOCKET_DATA, onSocketData);
    }
    protected function onConnected(event:Event):void {
      //make a GET request. HTTP requests end in two newlines.
      var request:String = "GET /files/hand.xml HTTP/1.1\n" +
                           "Host: actionscriptbible.com\n\n";
      sock.writeUTFBytes(request);
    }
    protected function onSocketData(event:ProgressEvent):void {
      //we got some data... let's just let it fill up the buffer
      //and wait for the server to disconnect!
    }
    protected function onClosed(event:Event):void {
      var response:String = sock.readUTFBytes(sock.bytesAvailable);
      trace(response);
    }
  }
}
```

This example is far from an actual HTTP client; it merely shows enough understanding of the HTTP protocol to make a request for a file, get at the data the server returns, and close the connection.

XML Socket Services

An XML socket server is just a socket server with a simple protocol: requests and responses are XML documents terminated by a `null` byte. Sure, you can send and receive XML through a web service, but web services use HTTP, so they include HTTP headers with each request and response. HTTP is

also a request-response model, so the server can't simply push a message to you without your asking. Furthermore, each request-response in HTTP is a full socket lifecycle, whereas an XML socket server maintains the same connection as long as necessary. These advantages, coupled with the ubiquity of XML and the simplicity of implementing it on the server as well as the client, make XML socket servers a good choice for low-latency, bidirectional communication, such as chat and multi-player games.

Many multiuser servers do XML sockets out of the box, including Swocket (`http://swocket.sf`
`.net/`), Palabre (`http://palabre.gavroche.net/`), SmartFoxServer (`http://www.`
`smartfoxserver.com/`), and ElectroServer (`http://www.electro-server.com/`). Alternatively, you can build on top of an existing socket server, adding on XML parsing to roll your own XML socket server.

For an ActionScript 3.0 client to connect to your XML socket server, you can easily build on `Socket`. However, Flash Player gives you an even simpler interface to XML socket servers with the `XMLSocket` class. In many ways, `XMLSocket` works identically to `Socket`; sending and receiving is simplified because the protocol is predetermined.

To communicate with an XML socket server, create a new `XMLSocket` object, `connect()` to the server, and listen for `Event.CONNECT` just as you would with a `Socket`. Sending data is now as simple as using the `send()` method with an `XML` parameter. Receive data by listening for a `DataEvent.DATA` event; the event object will contain the XML message in its `data` parameter.

```
var socket:XMLSocket = new XMLSocket();
socket.connect("example.com", 8080);
socket.addEventListener(Event.CONNECT, onConnect);
socket.addEventListener(DataEvent.DATA, onData);
private function onConnect(event:Event):void {
  var message:XML = <echo>ping</echo>;
  socket.send(message);
}
private function onData(event:DataEvent):void {
  var message:XML = XML(event.data);
  trace(message);
}
```

Flash Remoting

Flash Remoting is a powerful tool for client-server communication. It was originally created to interface Flash Player clients with application servers built by Adobe, such as ColdFusion and Flash Communication Server (now Flash Media Server). Now, with the specification opened by Adobe, it can be used to connect Flash Player to virtually any back end. Flash Remoting adapters have been created for server-side languages like PHP (AMFPHP, Zend AMF, WebORB for PHP, SabreAMF), Python (PyAMF, AmFast), Ruby (RubyAMF), Perl (AMF::Perl), Java (BlazeDS), Erlang (erlang-amf), as well as application frameworks that use these languages. Like all client-server technologies, at its core, Flash Remoting allows the client to invoke methods on the server and receive a response.

For the server, Flash Remoting provides a gateway that dispatches service requests to the proper code, so you don't have to handle incoming connections. The Remoting adapter invokes the proper server-side code depending on the request. In many implementations, the adapter maps service requests to classes and methods on the server side.

Flash Remoting uses a more structured protocol than the ones I've discussed so far in this chapter: the Action Message Format (AMF). This is a compressed, binary message format, which is usually sent over HTTP, but it can utilize other protocols as well. AMF has several benefits over other formats such as XML:

- Native type support — In XML, you have to interpret all data back from strings. AMF supports native data types on both the client and the server (where an appropriate type can be mapped to). This means that no parsing, conversion, or even casting is necessary for these types: `undefined`, `null`, `Boolean`, `int`, `Number`, `String`, `XMLDocument`, `Date`, `Array`, `Object`, `XML`, or `ByteArray`. You can use `IExternalizable` to serialize custom classes into AMF as well.

- Size — AMF packets are tightly packed, binary, and compressed. They take up far less bandwidth than the equivalent XML in all conceivable cases. This means you can send more data, more quickly.

- Speed — AMF is quickly interpreted by Flash Player, whereas larger XML documents can take a few moments to parse. Many of the server-side remoting adapters are also able to prepare AMF data with great speed.

The native type support in AMF works especially well when you pass complex objects back and forth between client and server. The remoting adapter allows these objects to come through the network barrier intact and in an equivalent representation for languages on both sides.

NetConnection

You use a `NetConnection` object to connect to a service that has a remoting gateway. This gateway is the URL through which service calls may be made. To set up a connection to a remoting gateway, use the object's `connect()` method with the remoting gateway URL:

```
var nc:NetConnection = new NetConnection();
nc.connect("http://server.com");
```

Note
`NetConnection` is also used for playing video and for integrating with Flash Media Server (FMS). FMS goes above and beyond Flash Remoting by providing object synchronization and server-to-client calls, as well as client stream publishing. Flash Media Server is not covered in this book, but you can consult the documentation available at `http://www.adobe.com/support/documentation/en/flashmediaserver/`. ■

You should use one `NetConnection` object per gateway that you would like to connect to. This is usually one, as you're more likely to call multiple methods on the same gateway than you are to have separate gateways. The `NetConnection` object is reused between different calls, even if internally it is creating and terminating multiple HTTP sessions.

Once you're "connected," make service calls with the `call()` method. The `call()` method of the `NetConnection` object invokes a server-side method. Its signature is

```
function call(command:String, responder:Responder, ... arguments):void
```

`command` is the name of the method. `responder` is a `Responder` object, covered in the next section, and (optionally) any arguments the method takes.

When you're finished working with a `NetConnection`, you should call the `close()` method to close the connection to the server. Any queued data that hasn't yet been sent to the server will be discarded, and any data that the server has yet to send to the application will not be received. To reconnect, you don't need to create a new `NetConnection` object; you can simply call `connect()` on an object previously created and go from there.

Responder

The `Responder` is an object that responds to the server's response to a `NetConnection` call. You pass it to `call()` to act as a delegate for the response from the server; it specifies what to do with the result and what to do in case of failure. The constructor for a `Responder` has one or two parameters: the function that will be called when the server returns a method, and the function to be called in case of an error. You can use a function reference — the name of a method in scope — just as well as an inline anonymous function, or a `Function` variable. Following is a code snippet that creates a `Responder` with appropriate handler methods:

```
var responder:Responder = new Responder(resultHandler, errorHandler);
function resultHandler(result:Object):void {
  trace("Success! Received result.");
  for (var property:String in result) {
    trace(property + " : " + result[property]);
  }
}
function errorHandler(error:Object):void {
    trace(error.description);
}
```

Notice that the parameters used by the result and error handler functions are not `Events` as they would be with normal event handlers. The result handler is passed whatever the server call returns; in this case, you've assumed it's an `Object` with properties. If the server-side method returns a single integer, the responder's result handler function should expect an integer parameter.

Summary

- You can use the `URLLoader` class to communicate with any HTTP-based service. Pass parameters via GET using the URL string or by via POST using the `data` property.

- SOAP and ad-hoc web services can simply use `URLLoader`, but SOAP and XML-RPC web services are eased by using a library.

- Sockets let you code lower-level protocols on top of TCP/IP rather than on top of HTTP. Use the `Socket` class and the I/O stream methods from `IDataInput` and `IDataOutput`.

- XML socket servers are servers that keep a persistent connection open to send and receive XML fragments. The client is notified whenever data has been sent from the server via the `XMLSockets` instance.

- Flash Remoting is a method of communicating with a server using AMF, which is faster and smaller than other client-server protocols.

Storing and Sending Data with SharedObject

When you're building programs in ActionScript that run inside a browser, you don't have unlimited access to the user's hard drive. At best, you can load or save one file at a time, with the user's permission in Flash Player 10, as you'll see in Chapter 30, "File Access."

However, using local shared objects, you can easily store all kinds of information that can be retrieved when the program runs again, even after the user shuts down the browser or her computer — usually without explicit user action. Using `SharedObjects`, you can give users the power to personalize your programs, retain those preferences, and improve their experience.

Comparing Approaches to Persistent Storage

There are several ways that SWFs playing in the browser can attain persistent storage. By *persistent storage*, I mean the ability to store data, close the browser, and then come back later and retrieve the data. All the approaches have different strengths and weaknesses, so let's compare them briefly.

Storing Information in a Local Shared Object

Using `SharedObjects` to persistently store data from ActionScript is as cheap, painless, and efficient as instant noodles for dinner, and probably better for you, too. However, `SharedObjects` have some limitations in terms of the size of the information you can store, and their dependence on the host computer.

The biggest limitation of `SharedObject` data is that it's associated with a single machine. When you store information in a local shared object, it is saved on the computer. When you use the program again from the same computer, you have access to that information; when you use it from

another user's account or another computer, you don't. On shared computers that use a single guest account rather than registered user accounts, like those in some libraries and schools, the shared objects' data is also shared. For instance, if you remember the user's name, address, and phone number on a public computer, there's a chance that the next user of the terminal has access to that information, presenting a clear privacy concern.

Storing Information in Local Files

In Flash Player 10 and later, and with the AIR runtime, you can write to and read from arbitrary files on the user's system. You can read more about this technique in Chapter 30. This is great for storing large amounts of data or making the output of a program portable so that the user can open it in other programs. In the Flash Player API, you rely on the user to provide the locations for both opening and saving files. Because of this restriction, local file access is not as convenient for saving preferences or repeated accesses to the same data. And, of course, this approach is not available in Flash Player 9.

Storing Information on a Server

If you're building a client-server application, you can always send things you need to be stored to the server. Ironically, though it requires sending the information over the network, this is the only approach for which you can guarantee some security. You can require authentication to get to information on a server, and you can transmit it across secure connections. Finally, unlike with files and shared objects on the user's machine, you can control the longevity of and guarantee the persistence of the information stored on it.

However, using a server to maintain information from a SWF means a lot of additional work. You must develop code on the server that can accept and retrieve the requested information. You must develop a way in which sessions can be tracked, and you can verify that the same user is retrieving his own information. You must determine and implement a way to serialize and deserialize information sent to and from the server.

A server that stores information can be made secure and can handle bigger loads of data, but it is far from the convenience of using a `SharedObject`.

Storing Information in the Browser

Web browsers allow a kind of persistent storage through cookies. Cookies are extremely limited in size, and they can only be string values, so if you want to store more complex types, you first have to encode them as strings. Because cookies are part of the browser's API in JavaScript, you have to interface with JavaScript code to use them. You can read more about interfacing with JavaScript in Chapter 43, "Interfacing with JavaScript."

Cookies are subject to the same limitations as `SharedObjects` as far as being associated with a physical computer and system account rather than with the user. A further limitation is that cookies are associated with the browser that created them. `SharedObjects` can store information for a SWF regardless of the version of Flash Player used to create them and which browser they were running in, but a cookie created while browsing in Firefox won't be available in Opera, even on the same computer. In addition, many savvy users clear their cookies frequently, so they may be more volatile than `SharedObject` storage.

On the other hand, the cookie mechanic may be desirable if you want users to be able to easily clear saved data. Because users are more likely to be savvy at clearing browser cookies than `SharedObject` data, placing your cached data in a cookie may be a concession to usability.

If you're interfacing with the browser, you may also have the ability to plug into other methods of offline storage, such as the Google Gears plug-in (`http://code.google.com/apis/gears/`) or HTML 5's web storage mechanism (`http://dev.w3.org/html5/webstorage/`). These more mature technologies provide database-like local storage and may be preferable in some cases. Of course, you'll still need a JavaScript bridge to interface with these browser-based technologies. Table 29-1 summarizes some of the issues inherent in using these four approaches to persistent storage.

TABLE 29-1

Solutions for Persistent Storage from ActionScript

	Local Shared Objects	Local Files	Cookies	Servers
Ease of Implementation in ActionScript 3.0	Easy	Easy	Intermediate	Intermediate to difficult
Additional Work to Use	None	None	Some	Some to lots
Privacy	Low	Low	Low	High
Persistence	Medium	Low	Low	High
Storing Typed Data	Easy	Easy	Difficult	Intermediate
Default Size Limit	100KB	100MB	4KB	None

Identifying Useful Situations for Shared Objects

Using local shared objects is great when you need to store small to medium amounts of non–mission-critical information that is not private. Local shared objects are excellent for storing users' preferences without requiring them to sign in. These are some ideas for how you can use shared objects:

- Store a user's progress in a game.
- Keep bookmarks in text, audio, and video, so that the user can come back to long media and continue right from where he left off.
- Remember the text size picked by the user, so that she can read text in a way that is comfortable to her.
- Save the user's preference to hear music or sound effects on a site that provides them.
- Retain any kind of theme customizations such as colors and background images.
- Remember which parts of your program have been used or visited.
- Present help or intro animations only once.

Returning to an application that remembers these kinds of small details without asking can be a positive experience. These are good examples because they are not personal or private; they help users in tangible ways when they have already expressed a preference that is not likely to change, and they are small features that it would be frustrating to authenticate just to retrieve.

Note

In Flash Player 10.1 and later, shared objects respect "private browsing" settings available on many browsers, so shared objects created during a private session will be wiped at the end of it. ■

Shared Objects and Remoting

It should be mentioned that `SharedObjects` have a secret second life. While during the day, the mild-mannered `SharedObject` helps you remember bits of information by keeping them filed away and synchronized on the user's hard drive, come nightfall it does a quick costume change and instead negotiates synchronized objects over the network with Flash Media Server and compatible servers. You call the first kind of `SharedObject` *local* shared object, and the second *remote* shared object. I'll focus on local shared objects in this chapter. Although the concepts for connecting and synchronizing with a remote shared object differ from those of loading and flushing to a local shared object, simple modifications of properties in shared objects are the same.

It's incredibly useful to be able to deal with real class instances on either side of a client-server application, rather than always converting from XML, JSON, or another transport format. Beyond being able to store typed data, AMF, the format used in shared objects, is binary, compressed, and fast. So, consider applying these serialization and deserialization strategies to support Flash Remoting in your client-server program.

For more information on remote shared objects, see Chapter 28, "Communicating with Remote Services," as well as the AS3LR entries on `NetConnection` and `SharedObject`, and the Flash Media Server developer documentation, available at `http://bit.ly/fms-dev-guide`.

Using SharedObjects

Using `SharedObjects` is about as simple as it can get. You retrieve a `SharedObject` and then read and write to it. Things you put in it, stay in it.

Retrieving a SharedObject

Before you can start storing information in a `SharedObject`, you must get one. Your program can create and use as many as it needs until it exceeds the space limitation set for it. More on that later, however. Because you can create lots of `SharedObjects`, you have to identify them by a unique name. To get a local `SharedObject`, use the static `SharedObject.getLocal()` method. The full class name is `flash.net.SharedObject`, so don't forget the import statement.

Caution

The name of the shared object cannot contain spaces or these characters: ~ % & \ ; : " ' , < > ? #. ■

In Example 29-1 you retrieve the `SharedObject` named `hiscores`. The first time you try to get the `SharedObject` with this name, it is created for you, so you don't have to worry if a `SharedObject` already exists.

```
import flash.net.SharedObject;
var hiscores:SharedObject = SharedObject.getLocal("hiscores");
```

Caution

Do not use new to construct a SharedObject. Use the static method SharedObject.getLocal(). ■

Reading from and Writing to a SharedObject

After you have retrieved a local shared object (LSO), use its data property, an Object, as an associative array. Any property you add to the data object is stored and available in later sessions.

You can set and retrieve properties of data, like any other Object, using either dot notation or array access notation. For a refresher on using Objects as associative arrays, see Chapter 10, "Objects and Dictionaries." To read from the SharedObject, just access the desired property of the data object. If you try to access a property that has not yet been stored, you get undefined.

EXAMPLE 29-1 http://actionscriptbible.com/ch29/ex1

Reading and Writing SharedObject Data

```
package {
  import com.actionscriptbible.Example;
  import flash.net.SharedObject;
  public class ch29ex1 extends Example {
    public function ch29ex1() {
      //retrieve or create a SharedObject
      var hiscores:SharedObject = SharedObject.getLocal("hiscores");

      //initialize properties if they don't exist
      if (!hiscores.data.highest) hiscores.data.highest = 999999;
      if (!hiscores.data.scores) hiscores.data.scores = new Array();

      //add a new entry
      hiscores.data.scores.push(
        {score: int(Math.random()*1000), time: (new Date()).time}
      );

      //dump all the data
      trace("HIGHEST SCORE: " + hiscores.data.highest);
      for (var i:int = 0; i < hiscores.data.scores.length; i++) {
        trace("Score:\t" + hiscores.data.scores[i].score +
              "\tat\t" + new Date(hiscores.data.scores[i].time).toString());
      }
    }
  }
}
```

Caution

You can't assign a value to the data object. You may add only properties to it. ■

And that's it! Every value you put into the `data` property of a `SharedObject` will be there when you retrieve it the next time with `getLocal()`. If you try reloading the previous example, you'll see the record of high scores grow with every access of the SWF.

Deleting Information from a SharedObject

Because the `data` property of a `SharedObject` is an `Object`, you can delete properties from a LSO just as you would delete them from an `Object`. Just use the `delete` operator with the property you want to clear from the `data` property, as the following snippet shows:

```
var so:SharedObject = SharedObject.getLocal("hiscores");
delete so.data.highest; //delete the top score
```

Removing properties from a `SharedObject` ensures that they won't be available the next time you attempt to retrieve them. Additionally, the space available to `SharedObjects` is fairly limited, so it's a good idea to clear out larger structures if they are no longer needed.

To totally wipe out the entire `SharedObject`, use its `clear()` method. This not only removes all the stored data, but the `SharedObject` file.

```
var so:SharedObject = SharedObject.getLocal("hiscores");
so.clear(); //nuke the LSO
```

After using `clear()`, the next time you request a local `SharedObject` with the same name, it won't be found and a new one will be created for you and returned.

Saving Information

You don't have to do anything to ensure that the local shared object's information is stored for later reference. An LSO is be synchronized to disk when the SWF is unloaded or when the `SharedObject` is no longer used and your program retains no more references to it.

However, you can force the `SharedObject` to save to disk manually by calling its `flush()` method. With no arguments, the method simply forces the LSO to write itself to disk. It's unlikely you'll ever need to do so, because the LSO saves automatically.

When passed an argument, the `flush()` method is useful for reserving space for large objects, so I will revisit it in "Working with Size Constraints."

Sharing Information between SWFs

By default, when you save information to a local shared object, it is associated with the SWF that created it. The SWF is identified by the location it was loaded from, so that each SWF gets its own whole namespace of possible `SharedObject` names. This means you don't have to worry if another SWF on the internet wants to use the name `storage` for its `SharedObject`.

To understand this better, let's lift back the hood of Flash Player and find how `SharedObjects` are stored on your local drive. Somewhere in your drive, under your username, is a folder that stores all

those `SharedObjects`. Let's call this directory `#SharedObjects`. When you create an LSO named `storage` in a file hosted at `http://example.com/examples/sharedobjects.swf`, information stored in that shared object ends up in a file named

```
#SharedObjects/example.com/examples/sharedobjects.swf/storage.sol
```

This literal mapping between the name of the `SharedObject` and a file on the filesystem explains why you are prevented from using specific characters when naming an LSO.

Note
I'm using the `#SharedObjects` alias for the shared object directory because the full path differs by operating system and contains a random string. If you poke around your filesystem, you can find this actual directory. For me it happens to be

```
C:\Users\roger\AppData\Roaming\Macromedia\Flash Player\#SharedObjects\LAHE8YH2
```

The Wikipedia article `http://en.wikipedia.org/wiki/Local_Shared_Object#File_locations` **can help you find your own shared object directory.** ∎

This scheme of storing shared object files also allows you to share shared objects between SWFs hosted in different locations on the same domain. You can, for example, store a central set of preferences that are implemented across several modules seamlessly.

To be able to share local shared objects, you must ensure that the shared object file is stored in a place that is accessible to SWFs in both locations. If you placed the shared object file at

```
#SharedObjects/example.com/storage.sol
```

it would be accessible to any SWF stored on `http://example.com/`. Likewise, storing it in the `examples/` directory would make it accessible to any SWF stored in the `examples` directory or its subdirectories, but not in other directories or other domains.

To modify the location of the local shared object, you can pass a second parameter to the `SharedObject.getLocal()` method. This optional parameter defines the path (relative to the domain) under which the local shared object file will be stored, and by extension it defines which SWFs can share data stored in it. The parameter defaults to the location of the SWF, as in the following example:

```
//in code run by http://DOMAIN/ORIGINAL/PATH/test.swf ...

SharedObject.getLocal("NAME");
//creates or retrieves #SharedObjects/DOMAIN/ORIGINAL/PATH/test.swf/NAME.sol

SharedObject.getLocal("NAME", "/");
//creates or retrieves #SharedObjects/DOMAIN/NAME.sol

SharedObject.getLocal("NAME","/NEW/PATH");
//creates or retrieves #SharedObjects/DOMAIN/NEW/PATH/NAME.sol
```

577

By targeting all your SWFs on the same domain to use a shared object file at the same location and choosing that location so that it is accessible to all your SWFs, you can share one set of data persistently between multiple programs or multiple parts of a program.

Caution

It is not possible to move the LSO file outside of the domain name directory. The highest directory you can set for the `localPath` parameter is /, which maps to the root of the domain. Using this value for `localPath` allows any SWF in the domain to access the `SharedObject`. Be aware that Flash Player's security sandbox restrictions still apply in addition to the `localPath` parameter. ■

Requiring a Secure Connection

You can add a further layer of security by creating local shared objects that are available only from SWFs served through a secured (HTTPS) connection. Pass `true` as the optional parameter `security` of `SharedObject.getLocal()` to retrieve a `SharedObject` that is only valid for SWFs served from secured connections.

This is the final parameter that `getLocal()` can accept, so let's review its full method signature:

```
function getLocal(name:String,
                  localPath:String = null,
                  security:Boolean = false):SharedObject
```

When you retrieve a `SharedObject` from a SWF served over HTTPS, a `SharedObject` is retrieved or created that can't be accessed from non-HTTPS locations. Attempting to retrieve a secure `SharedObject` from an HTTP connection returns `null`.

```
//from https://example.com/shortbread.swf
var so:SharedObject = SharedObject.getLocal("desserts", "/", true);
so.data.cookies = 100;

//from https://example.com/frozen/icecream.swf
var so:SharedObject = SharedObject.getLocal("desserts", "/", true);
trace(so.data.cookies); //100
so.data.icecreams = [{flavor: "chocolate", rating: 5}];

//from http://example.com/hot/pie.swf
var so:SharedObject = SharedObject.getLocal("desserts", "/", true);
//returns null

//from http://example.com/hot/pie.swf
var so:SharedObject = SharedObject.getLocal("desserts", "/");
trace(so.data.cookies); //undefined
trace(so.data.icecreams); //undefined
```

In the preceding example, the `desserts` `SharedObject` can exist both as a secured `SharedObject` and an insecure one. These two live independently, but SWFs served from insecure connections such as `pie.swf` don't have access to the secure versions.

Caution

Storing local shared objects as secured guarantees only the ability or inability of Flash movies to access the data in them. However, any information you store in them is stored unencrypted on the user's computer in an accessible directory. Again, do not use local shared objects to store sensitive or private information, especially without the user's consent. ■

Sharing with ActionScript 1.0 and 2.0 SWFs

ActionScript 3.0 uses the AMF3 format for serializing data to local shared objects. However, SWFs that use prior versions of ActionScript use AMF0. By default, this means that a SWF written in AS3 can read from shared objects saved by SWFs written in AS1 or AS2. But those older SWFs can't access shared objects saved by SWFs that use ActionScript 3.0.

To restore compatibility in both directions, you can explicitly instruct `SharedObjects` to use AMF0 or AMF3 when encoding data. To do so, use the static `SharedObject.defaultObjectEncoding` property or the instance variable `objectEncoding`.

If you want to pick one format to use for the entire program, just set `SharedObject.defaultObjectEncoding` once during your program's initialization. If you need more fine control, you can set the `objectEncoding` property on each `SharedObject` instance as you require.

These properties accept `uint`s. The value 0 represents AMF0, and 3 represents AMF3, but you should instead use the static properties `AMF0` and `AMF3` of `flash.net.ObjectEncoding`, as shown in the following snippet:

```
import flash.net.ObjectEncoding;
SharedObject.defaultObjectEncoding = ObjectEncoding.AMF0;
```

If you choose to use AMF0, types new to ActionScript 3.0, such as `ByteArray`, `int`, and `XML`, will be stored incorrectly. You also won't be able to work with custom classes that use `IExternalizable` (see "Storing Custom Classes") while using AMF0.

Working with Size Constraints

Ultimately, the user has control over whether your program can store data locally, and how much. Therefore, technically you can't rely on local shared objects for your application to work correctly. However, the vast majority of users have local shared objects enabled, and in reality, you can make this a requirement without shutting out more than a user or two. Your power users are far more likely to disable Flash Player entirely than specifically limit shared object use. The same limitations hold true for browser cookies, and there are a great number of web applications that require cookies. In fact, local shared objects have an advantage here because you have the ability to query the user for more space when you need it.

User Settings

You, and any Flash Player user, can control Flash Player's policies on local storage by accessing the Flash Player Settings Manager. The Settings Manager is an embedded control panel that you can find by accessing `http://adobe.com/go/settingsmanager`.
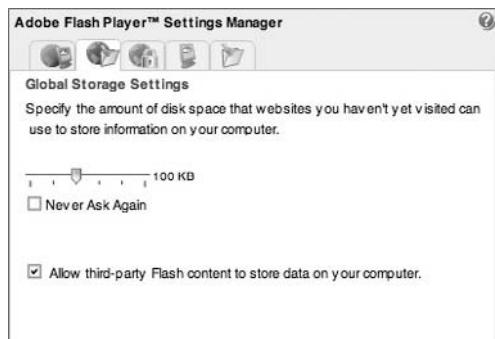
There are two relevant panels for controlling local shared object storage. You can set how much space is allotted by default for sites using the Global Storage Settings panel, as shown in Figure 29-1. You can control the space allotted to sites on a site-by-site basis, examine disk usage, and clear individual sites' local shared objects by using the Website Storage Settings panel, as shown in Figure 29-2.
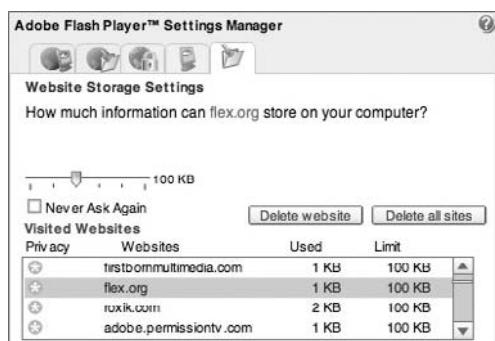
FIGURE 29-1

The Flash Player Global Storage Settings panel

FIGURE 29-2

The Flash Player Website Storage Settings panel

It's worth repeating that a majority of users will use your program with the default settings: 100KB allotted by default per site.

User Requests

When your program uses more than its allotment of space, rather than failing to save the requested information, the user is prompted to allow this action. Flash Player automatically opens a dialog box with this prompt, showing the space requested and the space currently allocated. You don't have to provide this dialog box yourself, but you also can't customize it. A sample dialog box is shown in Figure 29-3.

Caution

Any SWF that might display this kind of prompt must be at least 215 pixels wide by 138 pixels high or bigger to accommodate this prompt. If the stage is smaller, the request is automatically denied and the user sees no prompt! ■

Flash Player requesting additional space

This kind of prompting means that you can have the chance to store as much information as you want in a shared object, but also that it is never guaranteed. Furthermore, for some applications, allowing the user to see this dialog box might be unacceptable, and you might have to plan for a limit of 100KB and hope for the best.

Asking for Space before It's Too Late

Earlier you learned that `SharedObjects` are synchronized to disk automatically as the SWF unloads, so you don't have to use `flush()`. However, if your assignments to the LSO require it to ask the user to expand in size, it is too late to display this dialog box if the SWF is closing.

If you are using `SharedObjects` to casually store a tiny bit of noncritical information, this should be no problem and you may choose not to manually synchronize the shared object to disk with `flush()`. In this case, popping up a dialog box isn't even preferable to losing the information if the information is so unimportant.

However, if you need to know if the data was successfully stored, you should explicitly use `flush()`.

Asking for Space up Front

The `flush()` method can take an optional parameter that represents the disk space you anticipate needing. By using this parameter, you can limit user-facing dialog boxes to a single request.

For example, say you store 150KB, flush, store another 150KB, flush, and store a final 50KB and flush. With a default limit of 100KB, this results in three dialog boxes, first to expand from 100KB to 150KB, then to expand from 150KB to 300KB, and finally to expand from 300KB to 350KB.

Instead, you could — even before storing data to the `SharedObject` — flush while requesting 350KB of space. This presents a dialog box, but then (assuming the user accepts) the rest of the flush operations proceed without requiring user intervention, because the total space of 350KB was already allocated.

Using flush()

The signature of the `SharedObject`'s `flush()` method is

```
function flush(minDiskSpace:int = 0):String
```

Use the `minDiskSpace` argument to request the total space you anticipate needing, in bytes.

The return type lets you know if you need to wait for the user to accept or deny the request or if the operation failed. `flush()` can also throw an `Error` or dispatch an asynchronous `NetStatusEvent` error if the user specified to globally deny these requests or the dialog box can't be shown.

The possible return types are defined by static properties of the `SharedObjectFlushStatus` class:

- `SharedObjectFlushStatus.PENDING` is returned if the user has been presented with a dialog box requiring his approval.
- `SharedObjectFlushStatus.FLUSHED` is returned if the shared object has been successfully written to disk.

When calling `flush()`, you can check to see if user verification is necessary, and if so, subscribe to the `SharedObject` to be notified if the request is approved or denied. Example 29-2 shows how to do this by requesting space up front with the `minDiskSpace` parameter of `flush()`.

EXAMPLE 29-2 `http://actionscriptbible.com/ch29/ex2`

Asking the User to Expand SharedObject Space

```
package {
  import com.actionscriptbible.Example;
  import flash.events.NetStatusEvent;
  import flash.net.SharedObject;
  import flash.net.SharedObjectFlushStatus;

  public class ch29ex2 extends Example {
    protected var so:SharedObject;

    public function ch29ex2() {
      so = SharedObject.getLocal("storage");

      //request 1MB up front
      if (so.flush(1024 * 1024) == SharedObjectFlushStatus.PENDING) {
        so.addEventListener(NetStatusEvent.NET_STATUS, onUserAction);
        trace("User approval pending...");
      }
    }

    public function onUserAction(event:NetStatusEvent):void {
      so.removeEventListener(NetStatusEvent.NET_STATUS, onUserAction);
      switch (event.info.code) {
```

```
    case "SharedObject.Flush.Success":
      trace("Accepted");
      break;
    case "SharedObject.Flush.Failed":
      trace("Denied");
      //do error recovery
      break;
    }
   }
  }
}
```

Unlike most events in which the type of the event is communicated fully by its `type` parameter, the `NetStatusEvent.NET_STATUS` event can signal many kinds of events, so you must inspect the event's `info` object to determine what really happened.

Viewing Used Space

You can check the disk space already in use by a `SharedObject` by accessing its `size` implicit getter. This returns the size of the particular local shared object in bytes. However, it calculates this by measuring every object inside it, so it can be processor-intensive for exceptionally cluttered LSOs.

Storing Custom Classes

Using a `SharedObject`, you can store any native type in ActionScript 3.0 persistently. You can also store your own custom classes, with a little bit of extra work. The extra work has to do with the process of encoding objects into bytes so that they can be written to the local shared object file. Encoding objects into a flat binary representation is called *serialization*, and restoring them into their original form as runtime objects is called *deserialization*. You've already seen that Flash Player uses AMF to serialize and deserialize objects for storage in shared objects.

Storing Custom Classes without Modification

Certain kinds of classes can be serialized and deserialized in AMF with no problem. You can store and retrieve instances of a class in a `SharedObject` without additional code as long as:

- All properties of the class are public.
- The class's constructor takes no arguments, or all arguments to the constructor are optional.

Classes that are structured like this are essentially `Object`s with methods attached: AMF can serialize `Object`s, and methods are not serialized. Of course, all the object's properties must also be of serializable types for the object to be serialized.

The following is a custom class that can be natively serialized in AMF3. Because it uses an `int` type, you should use AMF3 as the encoding method. (Again, this is the default when programming in ActionScript 3.0.)

```
class Bookmark {
  public var page:int;
  public function Bookmark() {
  }
  public function toString():String {
    return "[Bookmark at page " + page + "]";
  }
}
```

Note that the Bookmark class has only public properties, and its constructor takes no arguments. Let's see how you can use and persistently store this custom class.

For Flash Player to know what class to associate an object with when it comes out of a serialized format, it must also know what class the bytes originally represented. So to use custom classes in a shared object, you must give the class a name that will be stored along with it. This must be done in both directions: to ensure that a name is saved along with a class's representation, and to know what class to create from the class's name when it is reconstituted. This is performed rather simply by one operation: registerClassAlias(). This function, in the flash.net package, registers a class definition with a String identifier. You must ensure that this is called before any SharedObject that contains custom classes is retrieved or created.

Then, to get the type back when retrieving an instance of a custom class, retrieve the property from the shared object's data property and upcast it to the type that it should be, as shown in Example 29-3.

EXAMPLE 29-3   http://actionscriptbible.com/ch29/ex3

Storing and Retrieving a Serializable Class in a SharedObject

```
package {
  import com.actionscriptbible.Example;
  import flash.net.SharedObject;
  import flash.net.registerClassAlias;
  public class ch29ex3 extends Example {
    protected var currentPage:int;
    protected var so:SharedObject;
    public function ch29ex3() {
      registerClassAlias("Bookmark", Bookmark);
      so = SharedObject.getLocal("storage");
      if (so.data.bookmark) {
        //go to the stored bookmark
        var b:Bookmark = Bookmark(so.data.bookmark);
        trace("found", b);
        gotoPage(b.page);

        //read the next page and bookmark it
        trace("advancing a page...");
        gotoPage(currentPage + 1);
        setBookmark();
      } else {
        //start at the first page if there's no bookmark
```

```
      gotoPage(1);
      setBookmark();
    }
  }
  public function setBookmark():void {
    var b:Bookmark = new Bookmark();
    b.page = currentPage;
    so.data.bookmark = b;
    trace("setting", so.data.bookmark);
  }
  public function gotoPage(page:int):void {
    trace("going to page", page);
    currentPage = page;
  }
 }
}

class Bookmark {
  public var page:int;

  public function Bookmark() {
  }

  public function toString():String {
    return "[Bookmark at page " + page + "]";
  }
}
```

To make sure there are no name collisions, it's customary to use the fully qualified class name as the alias name in `registerClassAlias()`. Another common trick to ensure that class aliases are registered before any code requiring them is run. To do this, set the result of your `registerClassAlias()` call to a static constant. This way, the function is run even before the constructor:

```
package {
  public class ch29ex3 {
    protected static const ALIAS:* = registerClassAlias("Bookmark", Bookmark);
  //...
```

The value `ALIAS` is never used; its purpose is simply to force `registerClassAlias()` to run. In fact, the function returns `void`; there is, in fact, no result to speak of.

Creating Self-Serializing Classes

You can get around the public variable restriction by using AMF3 and implementing the interface `flash.utils.IExternalizable` in your custom classes. By implementing this interface, you define how your class is turned into bytes and how it restores itself from bytes. In other words, if you elect to figure out how to do the serialization and deserialization of your class, Flash Player simply relies on the encoding methods you provide.

The IExternalizable interface includes two functions that you must implement:

```
function readExternal(input:IDataInput):void
function writeExternal(output:IDataOutput):void
```

These methods, when implemented, deserialize your class from, and serialize your class to, binary data. The IDataInput and IDataOutput interfaces are implemented by ByteArray; see Chapter 13, "Binary Data and ByteArrays" for more information on how to use these interfaces.

Internally, you can take advantage of AMF3 by using the readObject() and writeObject() properties of IDataInput and IDataOutput, which perform AMF3 serialization and deserialization on Objects. For instance, you might simply construct an Object of your nonpublic properties and use AMF3 to serialize that Object for you. This way, you don't have to worry exactly how your instance collapses into bytes; you can control the serialization at a higher level and use AMF encoding to do the grunt work. Note that when you're implementing IExternalizable, you're provided with IDataInput and IDataOutput instances as arguments; you can use any method that these interfaces provide to help you serialize and deserialize your data.

The revised Bookmark class shown in Example 29-4 shows this approach in action. Note that you can protect your properties this way, which is important to maintain encapsulation.

EXAMPLE 29-4 http://actionscriptbible.com/ch29/ex4

Making a Class Serializable Manually

```
package {
  import com.actionscriptbible.Example;
  import flash.net.SharedObject;
  import flash.net.registerClassAlias;
  public class ch29ex4 extends Example {
    protected static const RED:uint = 0xff0000;
    protected var currentPage:int;
    protected var so:SharedObject;
    public function ch29ex4() {
      registerClassAlias("Bookmark", Bookmark);
      so = SharedObject.getLocal("storage");
      if (so.data.bookmark) {
        trace("Found bookmark...");
        trace(so.data.bookmark);
      } else {
        so.data.bookmark = new Bookmark(345, RED, "b");
        trace("Saving bookmark...");
      }
    }
  }
}
import flash.utils.IDataOutput;
import flash.utils.IDataInput;
import flash.utils.IExternalizable;
class Bookmark implements IExternalizable {
  protected var page:int;
  protected var color:uint;
```

```
protected var name:String;
public function Bookmark(page:int = 0, color:uint = 0, name:String = null) {
    this.page = page;
    this.color = color;
    this.name = name;
}
public function getPage():int {
  return this.page;
}
public function readExternal(input:IDataInput):void {
  var props:Object = input.readObject();
  page = props.page;
  color = props.color;
  name = props.name;
}
public function writeExternal(output:IDataOutput):void {
  var props:Object = {page: page, color: color, name: name};
  output.writeObject(props);
}
public function toString():String {
  return "[Bookmark at page " + page + " color: 0x" + color.toString(16) +
         " name: " + name + "]";
}
}
```

The most important aspects of this example are that the `Bookmark` class hides its properties and provides a public interface as good object oriented design dictates, and it performs the custom serialization simply. More complex objects that contain non-base-types or deep data structures require you to put a bit more thought into your serialization and deserialization methods.

By implementing `IExternalizable` and providing your own serialization and deserialization methods, you can store and retrieve custom classes that contain complex types without breaking encapsulation.

Summary

- Using `SharedObject` for locally shared objects is painless in ActionScript 3.0.
- `SharedObjects` give you a decent amount of storage.
- Local `SharedObjects` are comparable to browser cookies but have more persistence, security, and size, and they require no bridge to JavaScript.
- Storing information on a server is, in most cases, more secure and more appropriate for media or other large data.
- `SharedObjects` are tied to an account on a computer rather than an actual user.
- You can share information in a `SharedObject` between SWFs in different locations on the same domain by constructing an appropriate `localPath` argument.

- You can create local shared objects that are used only in SWFs hosted from secure connections.

- The user ultimately has control over the available size for shared objects.

- When requesting more data than is available, the user is prompted.

- You can find out whether prompts to expand the available `SharedObject` size were successful.

- You can store custom classes in `SharedObject` instances with `registerClassAlias()` and, if necessary, `IExternalizable`.

File Access

F lash Player can help your users connect their content with the rest of the world. With a good file API, you empower users to share their video, images, audio, and documents; to generate or receive receipts, maps, and shopping lists; and even to edit, post-process, or mash up songs or videos.

Abilities of the File API

The file management abilities of Flash Player have evolved over time, so different versions of Flash Player support different operations. Uploading files to the web via an HTTP post and saving files to the user's system from an HTTP request are supported in Flash Player 9 and up. Opening local files, accessing their content as a `ByteArray`, and saving content from Flash Player directly into a file on the user's system are supported in Flash Player 10 and up.

The AIR runtime includes even more detailed control over files, such as a stream interface to files, file-browsing components, and scripted access to files. I won't cover AIR runtime features here.

Any time you deal with files in Flash Player, the user is prompted to select a file or destination with a standard system dialog box. At no time can Flash Player open or save a file on its own accord. Nor can ActionScript code browse through the filesystem at will, even if the user is doing just that. Control is passed entirely away from ActionScript while the user confirms file actions.

Introducing FileReference

All file access in Flash Player starts at the `FileReference` class. Not only can a `FileReference` represent a reference to a file on the user's system, providing access to its attributes and (sometimes) contents, but it controls asynchronous operations like saving and loading those files.

A `FileReference` doesn't start out associated with a given file. Typically, you create an empty `FileReference` and then, using methods it provides, you prompt the user for the actual file it should refer to. Once the user makes her choice, it dispatches an event, and the object is set.

Because a `FileReference` starts out empty, you can create your own `FileReference` objects with its argument-less constructor:

```
var fileToUpload:FileReference = new FileReference();
```

What you do next with the reference depends on which file operation you're interested in. If you were to upload a file or load its contents into memory, you'd prompt the user to pick an existing file first, by calling `browse()`. Then, once the `FileReference` is associated with a file, you'd call `upload()` or `load()`. If you were to download a file or save bytes in memory down to a file, you'd again prompt the user for a destination file, this time with the `download()` or `save()` methods.

Alternatively, if you want the user to be able to select many files at once, you can use a `FileReferenceList` object. Just instantiate the object, and again call `browse()`. Once the user chooses one or more files, her `FileReferences` are stored in the `fileList` parameter.

Choosing a File

Before you load or upload a file, you have to pick one out. As I just said, to select a file, create an empty `FileReference` and ask the user to associate it with a real file by calling `browse()`. When you call `browse()` with no parameters, it opens a new browse dialog box from which the user can select a file.

```
fileToUpload.browse();
```

By default, the browse dialog box displays all file types. If you want to simplify the file picker, you can show only files of the correct types by passing an array of `FileFilter` objects to `browse()`. For example, if the user just clicked a button to upload a photo, you might want to only show `JPEG`, `PNG`, `RAW`, `TIFF`, or `CR2` files. Or if you know the back end can only deal with `JPEGs`, great — only show `JPEGs`!

One `FileFilter` object describes each desired set of file types, not merely a single file type. You'd usually do this for a whole class of related file types. If you provide multiple `FileFilters`, the user will get his choice of classes of files. The `FileFilter` constructor takes the following parameters, which may also be changed after creation:

- `description` — A human-readable description of the class of files, like `"Source code files"` or `"Photos"`.

- `extension` — A list of file extensions that match this class of file, as a `String`. Include the wildcard and period in each extension, and separate each one with a semicolon, as in:

  `"*.jpg;*.jpeg;*.png;*.raw;*.tif;*.tiff;*.cr2"`

- `macType` — Optionally, a list of Mac four-character file type codes, again as a semicolon-delimited `String`. You can probably safely ignore this now. Years ago, OS X began using file extensions instead of the creator and type codes.

Here's an example that opens a browse dialog box with two options. One filter displays only `JPEG`, `GIF`, and `PNG` images, and the second filter displays only QuickTime movies.

```
      fileToUpload.browse([
        new FileFilter("Image Files", "*.jpg;*.gif;*.png"),
        new FileFilter("Quicktime Movies", "*.mov")
      ]);
```

The `FileReferenceList` class also defines a `browse()` method that uses identical syntax to the `FileReference` method of the same name. The only difference is that the browse dialog box opened from a `FileReferenceList` object allows the user to select more than one file.

Determining When a File Is Selected

Calling `browse()` causes the browse dialog box to open. However, it doesn't guarantee that the user will select a file. The browse dialog box has two buttons from which the user can select: Open and Cancel. If the user selects Cancel, the browse dialog box closes without selected files being sent to Flash Player. In that case, you should know better than to go ahead with any file operations that depend on the `FileReference`.

Both `FileReference` and `FileReferenceList` objects dispatch `Event.SELECT` and `Event.CANCEL` events when the user clicks the Open and Cancel buttons, respectively. Example 30-1 illustrates how these events work. When the user clicks the browse files text, the application launches a browse dialog box. Watch to see what the user will do.

EXAMPLE 30-1　`http://actionscriptbible.com/ch30/ex1`

Prompting the User to Select a File

```
package {
  import com.actionscriptbible.Example;
  import flash.events.*;
  import flash.net.*;
  public class ch30ex1 extends Example {
    public function ch30ex1() {
      var fileReference:FileReference = new FileReference();
      fileReference.addEventListener(Event.SELECT, onSelect);
      fileReference.addEventListener(Event.CANCEL, onCancel);

      var btn:TestButton = new TestButton(100, 25, "Browse");
      addChild(btn);
      btn.x = btn.y = 20;
      btn.addEventListener(MouseEvent.CLICK, function(event:MouseEvent):void {
        fileReference.browse([
          new FileFilter("All Files", "*"),
          new FileFilter("Images", "*.png;*.jpg")
        ]);
      });
      trace("\n\n\n");
    }
    private function onSelect(event:Event):void {
      var fileReference:FileReference = FileReference(event.target);
```

continued

591

EXAMPLE 30-1 *(continued)*

```
      var extension:String = fileReference.name.match(/\.\w+$/)[0];
      trace("So you like " + extension + "s, huh?");
    }
    private function onCancel(event:Event):void {
      trace("What, you don't like files?");
    }
  }
}
import flash.display.*;
import flash.text.*;
class TestButton extends Sprite {
  public var label:TextField;
  public function TestButton(w:Number, h:Number, labelText:String) {
    graphics.lineStyle(0.5, 0, 0, true);
    graphics.beginFill(0xa0a0a0);
    graphics.drawRoundRect(0, 0, w, h, 8);
    label = new TextField();
    addChild(label);
    label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
      false, null, null, "center");
    label.width = w;
    label.height = h;
    label.text = labelText;
    label.y = (h - label.textHeight)/2 - 2;
    buttonMode = true;
    mouseChildren = false;
  }
}
```

Did you notice? I went ahead a little bit and showed you properties of the file.

Retrieving File Properties

Once a `FileReference` object is associated with a real file, its accessors can give you a variety of information about the file it references, including

- `name` — The name of the file
- `size` — The size of the file in bytes
- `type` — The file extension in Windows and Linux, or four-character file type code in Mac OS X
- `creationDate` — A `Date` object for the time at which the file was created
- `modificationDate` — A `Date` object for the time the file was last modified
- `creator` — The creator type in Mac OS X

Caution

Calling any of these accessors results in an `IllegalOperationError` if the `FileReference` is not associated with a file.

On Mac OS X, the `type` parameter of a `FileReference` stores the file type code. This and the creator code are deprecated methods of determining file type used in Mac OS 9 and earlier. These values will be `null` more often than not. To determine the file type more reliably for modern files, look at the file's extension by manually parsing it out of the file's `name`, as in Example 30-1.

As an aside, the idea of "file type" is more tricky than you might realize, and Apple's current way of identifying it is the Universal Type Identifier. See `http://bit.ly/osx-utis` for lots more information. ■

Uploading a File

Once it's selected, you can upload a file via the `upload()` method of `FileReference`. If you used a `FileReferenceList` object to let the user select more than one file, you have to call `upload()` on each file independently.

The `upload()` method requires one parameter — a `URLRequest` object specifying the web service to which you want to upload the file. The `URLRequest` object must point to a resource that can handle HTTP file upload requests, such as a PHP script like the one used in the exercise in the following section. When the file data is uploaded, it is sent via HTTP POST with a content type of `multipart/form-data`. By default, the content name (by which you can reference the file data from the script) is `Filedata`. Therefore, to the server resource handling the upload, it is as though the file is being uploaded via an HTML form with a file input named `Filedata`. If you prefer to use a custom name, you can specify a second, optional parameter to the `upload()` method.

```
fileReference.upload(new URLRequest("upload.php"), "CustomFile");
```

The `upload()` method, like all network access in Flash Player, is asynchronous. `FileReference` objects dispatch events as you attempt to upload files, identical to the events dispatched by `URLLoader` while downloading a file. Similarly, uploading files is subject to the same security model as any other HTTP request from Flash Player. Either the URL must be in the same domain as the SWF, or the server must have a policy file. Review `URLLoader` events and networking security in Chapter 27, "Networking Basics and Flash Player Security."

One additional event is dispatched when uploading a file. Beyond `Event.COMPLETE`, which is dispatched when the upload is complete, the `FileReference` object also dispatches a `DataEvent.UPLOAD_COMPLETE_DATA` event if the server accepts and finishes your upload, and then it has something to say about it — if the web service has an HTTP response to your HTTP POST request. The response string is passed to you in the `data` property of the event object.

Adding Upload Handling to a Server

In this exercise, you build a simple application that lets users upload files to a server from Flash Player utilizing a PHP script on the server. To run the exercise, you'll need access to a web server that runs PHP — which is pretty much all of them — or a local installation of PHP, available for free at `http://php.net`, and a web server. (You can try one of the all-in-one easy installers like LAMP/WAMP/MAMP, which are totally painless, or you can use Web Sharing on Mac OS X, which should work out of the box.)

To get started, write a simple file upload script such as `simpleFileUpload.php`:

```php
<?php move_uploaded_file(
  $_FILES['Filedata']['tmp_name'],
```

```
            './'.time().$_FILES['Filedata']['name']
        );?>
```

Copy the PHP file to a hidden directory on your web host or to your local web server. It's not the most secure 100-character program, so for Pete's sake, delete it when you're done!

With the server-side code in place, you can run the client-side code in Example 30-2.

EXAMPLE 30-2   http://actionscriptbible.com/ch30/ex2

Uploading to a Server

```
package {
  import flash.display.Sprite;
  import flash.events.*;
  import flash.net.*;
  import flash.text.*;
  public class ch30ex2 extends Sprite {
    protected var fileRef:FileReference;
    protected var uploadButton:TestButton;
    protected var tf:TextField;
    protected const YOUR_UPLOAD_URL:String = "upload.php";
    public function ch30ex2() {
      fileRef = new FileReference();
      fileRef.addEventListener(Event.SELECT, selectHandler);
      fileRef.addEventListener(Event.CANCEL, cancelHandler);
      fileRef.addEventListener(ProgressEvent.PROGRESS, progressHandler);
      fileRef.addEventListener(IOErrorEvent.IO_ERROR, errorHandler);
      fileRef.addEventListener(SecurityErrorEvent.SECURITY_ERROR, errorHandler);
      fileRef.addEventListener(Event.COMPLETE, completeHandler);

      var btn:TestButton;
      btn = new TestButton(100, 25, "Browse...");
      btn.addEventListener(MouseEvent.CLICK, function(event:Event):void {
        fileRef.browse();
      });
      btn.x = btn.y = 20;
      addChild(btn);

      uploadButton = btn = new TestButton(100, 25, "Upload");
      btn.addEventListener(MouseEvent.CLICK, function(event:Event):void {
        fileRef.upload(new URLRequest(YOUR_UPLOAD_URL));
      });
      btn.x = 20; btn.y = 55;
      addChild(btn);

      tf = new TextField();
      tf.defaultTextFormat = new TextFormat("_sans", 11, 0);
      tf.multiline = tf.wordWrap = true;
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.width = 300; tf.x = 130; tf.y = 58;
      addChild(tf);
```

```
        cancelHandler(null);
      }
      protected function selectHandler(event:Event):void {
        tf.text = fileRef.name;
        uploadButton.mouseEnabled = uploadButton.tabEnabled = true;
        uploadButton.alpha = 1;
      }
      protected function cancelHandler(event:Event):void {
        tf.text = "";
        uploadButton.mouseEnabled = uploadButton.tabEnabled = false;
        uploadButton.alpha = 0.3;
      }
      protected function progressHandler(event:ProgressEvent):void {
        tf.text = "Uploading " +
          event.bytesLoaded + " / " + event.bytesTotal + "bytes ...";
      }
      protected function errorHandler(event:ErrorEvent):void {
        tf.text = event.text;
      }
      protected function completeHandler(event:Event):void {
        tf.text = "Upload complete!";
      }
    }
}
import flash.display.*;
import flash.text.*;
class TestButton extends Sprite {
  public var label:TextField;
  public function TestButton(w:Number, h:Number, labelText:String) {
    graphics.lineStyle(0.5, 0, 0, true);
    graphics.beginFill(0xa0a0a0);
    graphics.drawRoundRect(0, 0, w, h, 8);
    label = new TextField();
    addChild(label);
    label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
      false, null, null, "center");
    label.width = w;
    label.height = h;
    label.text = labelText;
    label.y = (h - label.textHeight)/2 - 2;
    buttonMode = true;
    mouseChildren = false;
  }
}
```

When you test the application, you should be able to browse to a file, select it, and then click the link to upload it. To verify that the file gets uploaded, change YOUR_UPLOAD_URL to point to a valid upload script you have control over, like the simpleFileUpload.php script I provided.

Downloading a File to Disk

Using the `download()` method of `FileReference`, you can save a file from a URL directly to a user's computer (of course, with a prompt). You can use this to bypass trying to explain how to save files in the browser, which might vary slightly between browsers and OSes. You can use it when a file is not ready immediately — as soon as Flash Player determines the file is ready for download, you can push the save dialog box at the user. Or, you can use this when a server generates dynamic content for the user, like a gift certificate, map, prescription, receipt, or questionable internet ministerial license.

Even though you don't call `browse()`, when the `download()` method is called, a new dialog box is opened prompting the user to save the file. The `download()` method requires one parameter: a `URLRequest` pointing to an HTTP resource to download:

```
var URL:String = "http://actionscriptbible.com/files/roger.gif";
fileToDownload.download(new URLRequest(URL));
```

Optionally, you can specify a second parameter, which determines the default name of the file as it appears in the download dialog box.

Because you're passing a `URLRequest`, not just a URL, you could put variables in the URL, as shown in Example 30-3. Check this out!

Downloading a File to Disk

```
package {
  import flash.display.Sprite;
  import flash.events.*;
  import flash.net.*;
  import flash.text.*;
  public class ch30ex3 extends Sprite {
    protected var fileRef:FileReference;
    protected var button:TestButton;
    protected var tf:TextField;
    public function ch30ex3() {
      button = new TestButton(100, 25, "Gimme");
      button.addEventListener(MouseEvent.CLICK, onGimme);
      button.x = 330; button.y = 20;
      addChild(button);

      tf = new TextField();
      tf.type = TextFieldType.INPUT;
      tf.defaultTextFormat = new TextFormat("_sans", 12, 0);
      tf.multiline = tf.wordWrap = false;
      tf.border = true;
      tf.width = 300; tf.height = 24; tf.x = 20; tf.y = 20;
      addChild(tf);
      tf.text = "TYPE HERE.";
    }
    protected function onGimme(event:MouseEvent):void {
```

```
        fileRef = new FileReference();
        //uses ENfer, a Smeltery free font - http://www.smeltery.net
        var URL:String = "http://actionscriptbible.com/files/text.php";
        var urlReq:URLRequest = new URLRequest(URL);
        urlReq.data = new URLVariables();
        urlReq.data.text = tf.text;
        fileRef.download(urlReq, "Your Dang Hot Pink Text.png");
        fileRef.addEventListener(Event.CANCEL, onReset);
        fileRef.addEventListener(Event.COMPLETE, onReset);

        button.mouseEnabled = button.tabEnabled = false;
        button.alpha = 0.5;
      }
    protected function onReset(event:Event):void {
        fileRef.removeEventListener(Event.CANCEL, onReset);
        fileRef.removeEventListener(Event.COMPLETE, onReset);

        tf.text = "";
        button.mouseEnabled = button.tabEnabled = true;
        button.alpha = 1;
      }
    }
}
import flash.display.*;
import flash.text.*;
class TestButton extends Sprite {
  public var label:TextField;
  public function TestButton(w:Number, h:Number, labelText:String) {
    graphics.lineStyle(0.5, 0, 0, true);
    graphics.beginFill(0xa0a0a0);
    graphics.drawRoundRect(0, 0, w, h, 8);
    label = new TextField();
    addChild(label);
    label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
      false, null, null, "center");
    label.width = w;
    label.height = h;
    label.text = labelText;
    label.y = (h - label.textHeight)/2 - 2;
    buttonMode = true;
    mouseChildren = false;
  }
}
```

Example 30-3 saves a file to your disk with the text you typed, rendered in a sweet hot pink font courtesy of PHP. Woo!

The sequence of events is like so. If the user clicks Cancel, the object dispatches an `Event.CANCEL` event. If the user clicks Save, the object dispatches an `Event.OPEN` event. As the file downloads, the object dispatches `ProgressEvent.PROGRESS` events just like `URLLoader`. And when the file has

downloaded, the object dispatches an `Event.COMPLETE` event. If there are errors, the appropriate error events are dispatched.

Loading a File into Memory

You can load files from your local system into Flash Player, without a round-trip to the server. In other words, even though you may be running a SWF from the internet, it can work off of your local files completely unplugged. You can load your photo into a toy or game, for instance, to customize your character. You can make a Papervision preview app that loads in local Collada files. Or if you're developing a game, you can write the editor with the ability to load and save levels, maps, and so on to your system. This ability is much more useful for offline applications, such as if you compile your SWF into a projector or you use AIR.

Version

FP10. Access to local files was added in Flash Player 10. In Flash Player 9, you can simulate this by roundtripping the file: upload the file to a server and then load it with a `URLLoader` to gain access to it. ∎

The contents of your file appear as a `ByteArray`, for you to play with in any way you see fit. Remember from Chapter 13, "Binary Data and ByteArrays," that with the power of byte-level access, as long as you understand the file format, you can interface with any kind of file in the world.

The process should be getting familiar by now. Simply select a file to load by creating a `FileReference` and calling `browse()`. Then load it into memory with `load()`, which is also asynchronous, broadcasting `Event.OPEN`, `ProgressEvent.PROGRESS`, and `Event.COMPLETE` events like any other load. Just no HTTP status event, of course. Although the file is available locally and should be able to load relatively quickly if it's on a hard drive, what if the "local" file is actually on a mounted network drive located halfway across the world? Making all the file operations work similarly helps keep the API uniform. One final note: if you attempt to load a file so large that Flash Player can't allocate enough memory to hold it, Flash Player throws a `MemoryError`.

Once the file is completely loaded, its contents are assigned to the `FileReference`'s `data` property as a `ByteArray`, as shown in Example 30-4. Couldn't be easier!

EXAMPLE 30-4 `http://actionscriptbible.com/ch30/ex4`

Loading a Local File into Memory and Using It

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.geom.Matrix;
  import flash.net.*;
  public class ch30ex4 extends Sprite {
    protected var fileRef:FileReference;
    public function ch30ex4() {
      //Click the stage to load in a pattern
      fileRef = new FileReference();
      fileRef.addEventListener(Event.SELECT, onFileSelect);
```

```
      stage.addEventListener(MouseEvent.CLICK, onChooseFile);
    }
    protected function onChooseFile(event:MouseEvent):void {
      fileRef.browse([new FileFilter("Image Files", "*.png;*.jpg;*.gif")]);
    }
    protected function onFileSelect(event:Event):void {
      fileRef.removeEventListener(Event.SELECT, onFileSelect);
      stage.removeEventListener(MouseEvent.CLICK, onChooseFile);

      fileRef.addEventListener(Event.COMPLETE, onFileLoad);
      fileRef.load();
    }
    protected function onFileLoad(event:Event):void {
      fileRef.removeEventListener(Event.COMPLETE, onFileLoad);

      //interpret the bytes into a DisplayObject
      var image:Loader = new Loader();
      image.loadBytes(fileRef.data);
      image.contentLoaderInfo.addEventListener(Event.COMPLETE, onImageParse);
    }
    protected function onImageParse(event:Event):void {
      //get the (uncompressed) bitmap that decoding the file results in
      var content:DisplayObject = LoaderInfo(event.target).content;
      var bitmapData:BitmapData = Bitmap(content).bitmapData;
      var m:Matrix = new Matrix();
      m.identity();
      m.scale(0.25, 0.25);
      graphics.clear();
      graphics.beginBitmapFill(bitmapData, m, true, true);
      graphics.drawRect(0, 0, stage.stageWidth, stage.stageHeight);
    }
  }
}
```

In this example, you load an image file locally and, assuming it's a format that Flash Player can understand, use Loader to get a raw bitmap out of the encoded file. Then you fill the screen with that image, tiled.

Saving Data to Disk

Last but far from least is the ability to store data from Flash Player right on your filesystem. This is great for applications that generate media in code. For example, the ActionScript PDF library AlivePDF (http://alivepdf.bytearray.org/) can generate a PDF file in memory and then prompt the user to save it to her system using FileReference. As you may have noticed, there are more ActionScript 3.0 libraries out there than you can shake a stick at, and a good number of them help you read and write files of some crazy format or other. Most of the application ideas discussed in "Loading a File into Memory" benefit from a symmetrical capacity to write out files to disk. The two go hand in hand.

Version

FP10. Access to local files was added in Flash Player 10. In Flash Player 9, you can upload the file to a server and then download it again to save it. ∎

The general procedure for saving a file should be no surprise by now. With an empty `FileReference`, call `save()`, passing it the data to write to a file, and optionally a default filename. (The user can always choose his own.) The OS save dialog box is presented to the user automatically, and events are broadcast that let you know what happened. If the user cancels the dialog box, `Event.CANCEL` is broadcast and nothing is saved. Otherwise, `Event.SELECT` is broadcast; then `Event.OPEN` is broadcast when the file is opened for writing, `ProgressEvent.PROGRESS` as it is being written to, and `Event.COMPLETE` when the file is done writing. Any write errors raise an `IOErrorEvent.IO_ERROR` event.

The `save()` method is a little more intelligent than the `load()` method, because Flash Player knows more about the kind of data you have in a typed ActionScript variable than it's willing to wager about the binary data it finds in a file. Depending on what type you pass to `save()`, Flash Player will help you out:

- `String` — Pass in text, and it will be written to a file as plaintext (using UTF-8 encoding).
- `XML` — Pass an XML object, and XML source will be written into the file.
- `ByteArray` — Binary data will be written to the file verbatim.
- `*` — Any other type of parameter will be converted to a string using its `toString()` method and written out as plaintext.

In Example 30-5, you create a world-class fully featured office productivity tool.

EXAMPLE 30-5 `http://actionscriptbible.com/ch30/ex5`

Loading and Saving Local Text Files

```
package {
  import flash.display.Sprite;
  import flash.events.*;
  import flash.net.*;
  import flash.text.*;
  public class ch30ex5 extends Sprite {
    protected var tf:TextField;
    protected var fileRef:FileReference;
    public function ch30ex5() {
      var btn:TestButton;
      btn = new TestButton(100, 25, "Load");
      btn.addEventListener(MouseEvent.CLICK, onLoadClick);
      btn.x = btn.y = 20;
      addChild(btn);
      btn = new TestButton(100, 25, "Save");
      btn.addEventListener(MouseEvent.CLICK, onSaveClick);
      btn.x = 130; btn.y = 20;
      addChild(btn);

      tf = new TextField();
      tf.type = TextFieldType.INPUT;
```

```
      tf.defaultTextFormat = new TextFormat("_sans", 12, 0);
      tf.multiline = tf.wordWrap = tf.border = true;
      tf.width = stage.stageWidth - 40;
      tf.height = stage.stageHeight - 75;
      tf.x = 20; tf.y = 55;
      addChild(tf);
    }
    protected function onLoadClick(event:Event):void {
      fileRef = new FileReference();
      fileRef.browse([new FileFilter("Text files", "*.txt"),
        new FileFilter("All Files", "*")]);
      fileRef.addEventListener(Event.SELECT, onLoadSelect, false, 0, true);
    }
    protected function onLoadSelect(event:Event):void {
      fileRef.load();
      fileRef.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onLoadComplete(event:Event):void {
      tf.text = fileRef.data.readUTFBytes(fileRef.data.length);
    }
    protected function onSaveClick(event:Event):void {
      fileRef = new FileReference();
      fileRef.save(tf.text, "ch30ex5.txt");
    }
  }
}
import flash.display.*;
import flash.text.*;
class TestButton extends Sprite {
  public var label:TextField;
  public function TestButton(w:Number, h:Number, labelText:String) {
    graphics.lineStyle(0.5, 0, 0, true);
    graphics.beginFill(0xa0a0a0);
    graphics.drawRoundRect(0, 0, w, h, 8);
    label = new TextField();
    addChild(label);
    label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
      false, null, null, "center");
    label.width = w;
    label.height = h;
    label.text = labelText;
    label.y = (h - label.textHeight)/2 - 2;
    buttonMode = true;
    mouseChildren = false;
  }
}
```

In the example, text can be loaded from a file into the text field, edited, and saved back into a file. Eat your heart out, Word.

Summary

- Flash Player 9 allows for file upload and download between your filesystem and a web server. Flash Player 10 allows files to be loaded and saved between memory and disk locally.

- The user must approve all file operations using a standard operating system dialog box. Flash Player can't access files without the user's explicit action. AIR can.

- Using the `FileReference` or `FileReferenceList` classes, you can prompt users to browse to and select a file or files from their local drive to upload or load files.

- With byte-level access and a file API, there's no limit to what you can create.

Part VII

Sound and Video

Playing and Generating Sound

S ound is a powerful tool. A little bit of well-chosen sound goes a long way in adding depth and character to your Flash application. In this section, I look at ways to integrate sound into your program.

Through the past few versions, Flash Player has iterated on its sound support, making it far more robust. You can play SWF-embedded sound and load and stream MP3s. You can mix sounds, pan, and fade them. You can get a spectrum of the audio coming out of Flash Player for audio visualization. In Flash Player 10 and above, you can even sample sounds and generate your own dynamically. I'll cover all these in this chapter.

Flash Player also gives you access to audio coming in from the user's microphone or other audio input. This is covered in Chapter 33, "Capturing Sound and Video."

An Overview of the Sound System

I'll tackle the sound system task by task. Before you start playing sounds, however, let's take a quick birds-eye view of the sound architecture in Flash Player. A few classes work in concert, handling different aspects of playing and mixing sounds. All these classes belong to the `flash.media` package.

- `Sound` — Represents sound data. When you're going to play sounds in Flash Player, you always start with a `Sound` object. Instances of this class hold, load, buffer, and accept audio data. Like the `Loader` class, `Sound` encapsulates the loading and decoding of files over the network. If you only need to play simple sounds, you can get away with only using this class. After you start playing a `Sound`, however, playback is controlled through a new `SoundChannel` instance that's created.
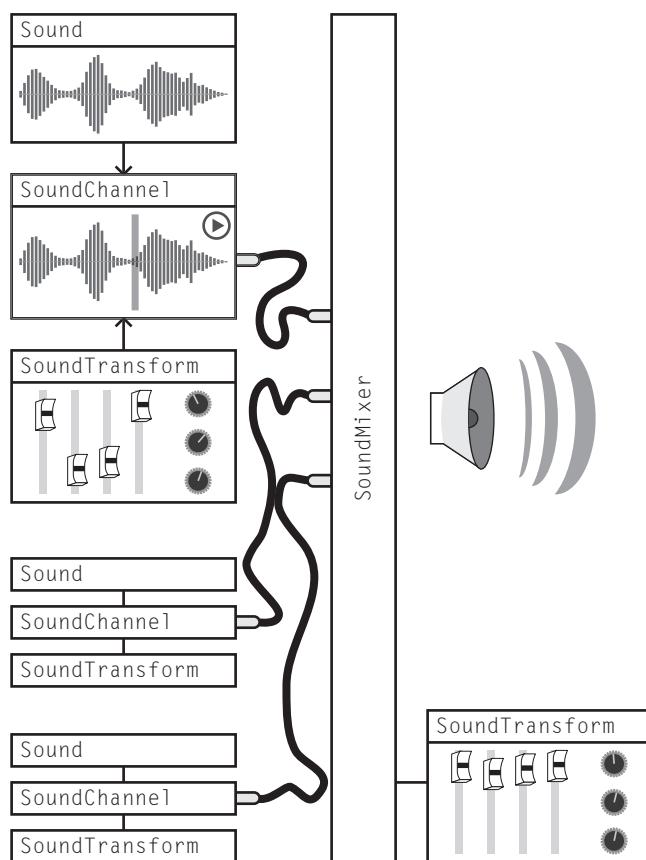
- SoundChannel — One audio stream that Flash Player mixes into its audio output. Represents an actively playing sound. Every time you play a Sound object, you get a new SoundChannel. Even if the audio is stereo — with two channels — it counts as a single SoundChannel. Flash Player mixes up to 32 simultaneous SoundChannels to produce the mix that it sends to your computer (which probably does its own additional mixing with other open programs and so on, but let's ignore that). Because it represents an actively playing audio stream, you can terminate it and control how it is mixed. You do this through a SoundTransform object.

- SoundTransform — Controls how a sound is mixed. Use its properties to change the volume and panning of the associated sound.

- SoundMixer — The global mix of all playing sounds in Flash Player. You can control the overall volume and panning of the mixed-down audio by modifying the SoundMixer's SoundTransform. Because this is a global object, it exposes all static properties and methods.

FIGURE 31-1

Classes involved in the Flash Player sound system

The relationship between these classes is shown in Figure 31-1. It's most important to understand the difference between Sound and SoundChannel. You have one Sound for each audio file and one SoundChannel for every currently playing stream. Say you have a game with a laser sound and an explosion sound: that's two unique Sound objects. When you destroy the final boss, you might hear 20 copies of the explosion sound layered on top of each other: that's 20 SoundChannel objects created from the same Sound object.

Sounds in Flash Player use the event framework to broadcast information about their current status. Events dealing with loading sounds, generating audio, and getting metadata from sound files are broadcast by Sound objects. An event specifying that the sound is finished playing is dispatched by a SoundChannel object.

Prepping and Playing Sound Objects

The audio data associated with Sound objects in Flash Player can come from different sources — either from a local file in your asset folder or from a remote URL, or it can be embedded into your SWF file. You can also generate audio data on the fly, which will be covered in the section "Synthesizing Audio."

Loading a Sound from an External File or URL

Besides holding onto audio data, the Sound class loads this data. While loading audio files, Sound instances behave much like Loader, covered in Chapter 27, "Networking Basics and Flash Player Security." Loading a sound from a URL is simple, whether your URL points to a local file or a file on the network, whether it is a relative or absolute path. Simply create the proper URLRequest and pass this to the Sound object's load() method. The signature of the load() method is

```
function load(stream:URLRequest,
              context:SoundLoaderContext = null):void
```

The required property stream locates an MP3 file to play. MP3 is the only format that Sound can load from a file. The optional context argument provides options for loading the file, similar to a LoaderContext. However, SoundLoaderContext also lets you set the buffer time: how much of the sound is loaded before you can start streaming it. Because the examples communicate across domains, I use this argument, but in most cases, you can leave out the context.

After your sound is loaded, playing it back is a piece of cake — just call play(). Example 31-1 shows how this works.

EXAMPLE 31-1 `http://actionscriptbible.com/ch31/ex1`

Loading and Playing a Sound

```
package {
  import flash.display.Sprite;
  import flash.media.Sound;
  import flash.media.SoundLoaderContext;
  import flash.net.URLRequest;
  public class ch31ex1 extends Sprite {
```

EXAMPLE 31-1 *(continued)*

```
    public function ch31ex1() {
      //The Four Seasons: Winter [performance & recording in the public domain]
      //composed by Antonio Vivaldi, performed by the US Air Force Band
      var winterSong:Sound = new Sound();
      winterSong.load(
        new URLRequest("http://actionscriptbible.com/files/winter.mp3"),
        new SoundLoaderContext(5000, true)
      );
      winterSong.play();
    }
  }
}
```

Caution

All the rules for Flash Player Security apply to the Sound object. Please reference the Flash Player security model introduced in Chapter 27. ∎

Buffering a Streaming Sound

When you play a sound before it is finished loading, Flash Player automatically buffers the sound, making sure that a certain amount of audio is loaded before actually playing, regardless of when you ask the sound to play. This is the case in Example 31-1. Because you call `play()` immediately after `load()`, the file hasn't even begun to arrive when you call `play()`. Because Flash Player automatically buffers the file, however, you needn't worry about that.

On slower connections, you can help keep the client from running out of audio to play by setting a longer buffer time. This value is specified in the `SoundLoaderContext` object. You can omit this parameter to use the default value of 1000ms (one second), or you can customize it like in Example 31-1.

Embedding Sounds in a SWF

You can include sounds directly in your SWF by embedding them, eliminating the need to load them at runtime. Additionally, when embedding sounds you have a much wider option of formats for the source file, and SWFs can embed multiple sounds per file. Perhaps most importantly, embedded sounds can be stored and played back using other audio codecs that Flash Player supports beyond MP3. These advantages make embedding sounds an attractive option.

You can embed sounds in the same SWF as your ActionScript code or in an external SWF. Sounds embedded in a SWF exist as subclasses of the `Sound` class so that you can use them easily. You can use these embedded `Sound` subclasses in the same way as other `Sound` instances, except of course you don't have to call `load()`.

Although embedding assets has been covered in Chapter 16, "Working with DisplayObjects in Flash Professional," let's look at specific options you have when embedding sounds. I'll pay particular attention to the supported source file formats and encoding formats. The technique, as always, differs whether you're compiling with Flash Professional or a Flex SDK-based toolchain (including Flash Builder).

Embedding Sound with Flash Professional

Of all methods, Flash Professional supports the most diverse input file formats and audio codecs, as well as the best control over transcoding quality. Table 31-1 lists the kinds of file that you can import. To embed sounds in a SWF with Flash and make them available to ActionScript, import the file to the Library as a Sound symbol, and assign it a linkage class name.

TABLE 31-1

Sound Files Supported for Import in Flash Professional

Format	Platform/s	Other requirements
ASND	Win/Mac	
WAV	Win/Mac	QuickTime must be installed on Mac.
MP3	Win/Mac	
AIFF	Win/Mac	QuickTime must be installed on Windows.
AU	Win/Mac	QuickTime must be installed.
QuickTime movies	Mac/Win	QuickTime must be installed.
Sound Designer II	Mac	QuickTime must be installed.
System 7 sounds	Mac	QuickTime must be installed.

To embed a sound in a SWF with Flash and make it available to ActionScript, import the file to the Library as a Sound symbol, and assign it a linkage class name.

1. Import the sound to the Library with File ➪ Import ➪ Import to Library.
2. Locate the sound symbol in the Library panel, select it, and choose Properties in the context menu. This opens the Sound Properties dialog, pictured in Figure 31-2.
3. Make sure the Export for ActionScript box is checked.
4. Give the sound a fully qualified class name; the class you specify here becomes a subclass of `Sound`

The Sound Properties panel also lets you customize how the sound is embedded by selecting a codec and customizing encoding options. This gives you excellent control over compression, letting you tweak the relationship between quality and file size. You can use the Test button to check the quality of the compressed audio. Encoding audio with Flash Professional is the only way to utilize the other audio codecs built into Flash Player beyond MP3.

The Sound Properties dialog, which provides sound export options
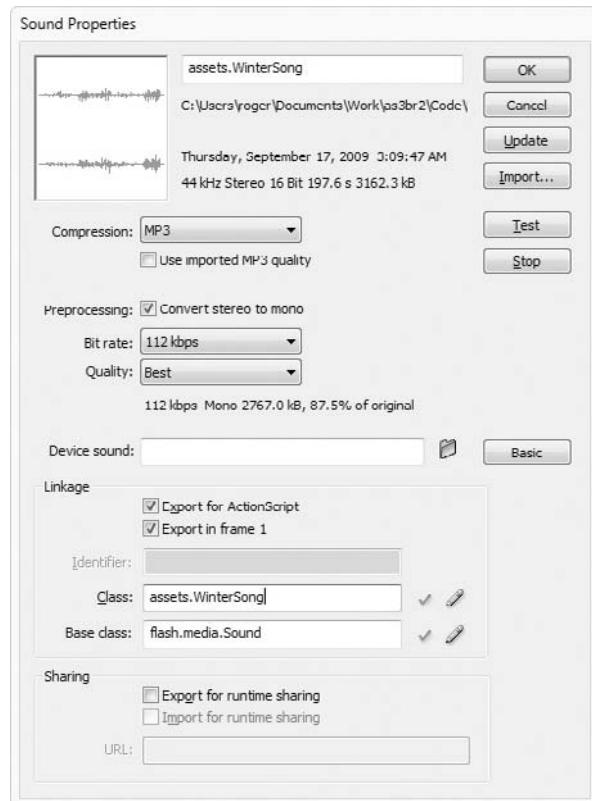
Table 31-2 compares the audio codecs Flash Player contains. Some of these codecs have special purposes. Only codecs with "Flash Pro" listed as an encoder can be embedded with Flash Professional.

Embedding Sound with Flash Builder

When using Flash Builder and other Flex SDK-based IDEs, you can embed sound data with the `Embed` metadata tag, which requires the location of the audio's source file. The Flex SDK compilers can only import and encode MP3s. Here's how you'd embed "Winter" from Vivaldi's *Four Seasons* into the SWF as the `Sound` subclass `WinterSong`.

```
[Embed(source="winter.mp3")]
public var WinterSong:Class;
```

The Flex SDK compilers don't provide encoding options with the `Embed` tag. The original MP3 data is used without transcoding.

TABLE 31-2

Audio Codecs Supported for Embedded Sound Playback in Flash Player

Codec	Notes	Encoders	Decoders
MP3	All-purpose, can be tuned for high fidelity as well as high compression.	Flash Pro, Flex SDK	Flash Player 9+
Uncompressed ("Raw")	You'd be pretty insane to use this.	Flash Pro	Flash Player 9+
ADPCM	Use for shorter clips.	Flash Pro	Flash Player 9+
Nellymoser ("Speech")	Use for speech.	Flash Pro, Flash Player 9+	Flash Player 9+
Speex	Use for speech; only produced when recording from Microphone in FP.	Flash Player 10+	Flash Player 10+
HE-AAC	Embedded in video streams.	N/A	Flash Player 9.0.115

Accessing Embedded Sounds

Regardless of which method you use to embed sound, to utilize the embedded sounds in your code, simply instantiate the class that the sound is embedded in to create a new `Sound` object. In this case, you're using a sound whose class was called `WinterSong`. This class, when initialized, already holds the proper audio data.

```
//if you're pulling the Class definition dynamically, use something like:
var WinterSong:Class = Class(getDefinitionByName("assets.WinterSong"));
var sound:Sound = new WinterSong();
sound.play();
```

Controlling Sound Playback

After you've fired off an audio stream from a `Sound` object, you can control its playback. Remember that the handle for an active audio channel is the `SoundChannel` object returned by calling `play()` on a `Sound`.

Playing and Stopping

If you're sick of the sound you've chosen to `play()`, you can `stop()` it just as easily. Just call the `stop()` method on the `SoundChannel` object. This stops the audio and resets its playhead to zero — rewinds it. Example 31-2 modifies the Vivaldi player to stop playing when it detects a click.

EXAMPLE 31-2 http://actionscriptbible.com/ch31/ex2

Stopping Playback

```
package {
  import com.actionscriptbible.Example;
  import flash.events.MouseEvent;
  import flash.media.*;
  import flash.net.URLRequest;

  public class ch31ex2 extends Example {
    protected var activeSound:SoundChannel;
    protected var winterSong:Sound;
    public function ch31ex2() {
      //The Four Seasons: Winter [performance & recording in the public domain]
      //composed by Antonio Vivaldi, performed by the US Air Force Band
      winterSong = new Sound();
      winterSong.load(
        new URLRequest("http://actionscriptbible.com/files/winter.mp3"),
        new SoundLoaderContext(5000, true)
      );
      stage.addEventListener(MouseEvent.CLICK, onClick);
      activeSound = winterSong.play();
      trace("Playing song...");
    }
    protected function onClick(event:MouseEvent):void {
      if (activeSound) {
        activeSound.stop(); //it's that easy
        activeSound = null;
        trace("Stopped!");
      }
    }
  }
}
```

After a SoundChannel is stopped, it's pretty much finished, and you can discard it. If you want to replay the sound, call play() on the Sound again, which creates a new SoundChannel.

If you are streaming an audio file, you can also stop it from playing back by using the Sound object's clear() method. This not only causes the sound to stop, but it stops the download of the streamed audio file. If you want to play the sound again, you have to reinitiate the loading of the sound file before you can play it.

In some cases, you'll want to quickly stop all audio playback. To handle this, you can use the nuclear option: SoundMixer.stopAll(). When you call this static method, all currently playing sounds are stopped immediately.

Seeking and Looping

You've used the `play()` method several times now, but I haven't explored its parameters. By using the optional parameters of `play()`, you can do much more with a sound. The full signature for `play()` is

```
function play(startTime:Number = 0,
              loops:int = 0,
              sndTransform:SoundTransform = null):SoundChannel
```

Let's look at each of these parameters:

- `startTime` — Determines the position of the playhead when the sound starts playing, in milliseconds. The default is 0; when you call `play()` with no arguments, the sound starts playing from the beginning.

  ```
  sound.play(2500); //plays the sound starting at 2.5 seconds in
  ```

- `loops` — Determines how many times the sound plays. The default is again 0, which actually plays the sound once. Values higher than 1 loop the sound when it reaches its end. Because this argument is an `int` you can't pass infinity, but you can pass `int.MAX_VALUE` if you want the sound to loop *almost* forever.

  ```
  beersOnTheWallSong.play(0, 99); //play the song 99 times
  ```

- `sndTransform` — Sets an initial `SoundTransform` to modify how the sound is mixed. Can also be set after the sound starts playing. I'll discuss how to use sound transforms in the section "Applying Sound Transformations."

All sound playback is achieved by sending different parameters to the `play()` method. I'll create some specific examples in the following section.

Fast-Forwarding, Rewinding, Pausing, and Restarting a Sound

In this section, you build a simple audio player. With just the `startTime` parameter of the `play()` method and the `position` property of the `SoundChannel` class, you can create all the standard audio player controls you know and love. I've already beaten `play()` and `stop()` to death with a stick.

With fast-forward or rewind, the goal is to jump forward or backward from the currently playing portion of the sound. To do this, you use the `position` property to get the current playhead position, add or subtract some value to that, and then restart the song from the new position.

Pausing a sound is almost the same. Because the `stop()` method causes the position to be lost, you just remember the playhead position at the time the user pauses. Later, when the song is resumed, you can use the value to pick up where you left off.

Example 31-3 a simple sound player example with four buttons — rewind, pause, play, and fast-forward — that uses the position parameter in the `play()` method. There is also a text field that shows the current position and the overall time in seconds.

Creating Standard Audio Controls with Sound Methods

```
package {
  import flash.display.Sprite;
  import flash.net.URLRequest;
  import flash.events.*;
  import flash.media.*;
  import flash.text.*;
  public class ch31ex3 extends Sprite {
    protected var sound:Sound;
    protected var channel:SoundChannel;
    protected var lastPosition:Number;
    protected var positionTF:TextField;
    protected var isPaused:Boolean = false;
    protected const SEARCH_RATE:int = 2000;
    public function ch31ex3() {
      createChildren();
      var songurl:String = "http://actionscriptbible.com/files/winter.mp3";
      sound = new Sound(new URLRequest(songurl));
      play();
    }
    protected function createChildren():void {
      var txtFormat:TextFormat = new TextFormat("_typewriter", 14, 0);
      var controlsTF:TextField = new TextField();
      controlsTF.defaultTextFormat = txtFormat;
      controlsTF.selectable = false;
      controlsTF.htmlText = '<a href="event:rw">&lt;&lt;</a>\t' +
        '<a href="event:pause">||</a>\t<a href="event:ff">&gt;&gt;</a>';
      controlsTF.addEventListener(TextEvent.LINK, onControlClicked);
      addChild(controlsTF);
      positionTF = new TextField();
      positionTF.defaultTextFormat = txtFormat;
      positionTF.x = controlsTF.textWidth + 15;
      addChild(positionTF);
      this.addEventListener(Event.ENTER_FRAME, updatePositionDisplay);
    }
    public function play(position:int = 0):void {
      if (channel) channel.stop();
      channel = sound.play(position);
    }
    public function rewind(event:Event = null):void {
      if (!channel) return;
      isPaused = false;
      lastPosition = channel.position - SEARCH_RATE;
      lastPosition = Math.max(0, lastPosition);
      play(lastPosition);
    }
    public function fastForward():void {
      if (!channel) return;
      isPaused = false;
      lastPosition = channel.position + SEARCH_RATE;
```

```
        lastPosition = Math.min(sound.length, lastPosition);
        play(lastPosition);
      }
      public function pause():void {
        lastPosition = channel.position;
        channel.stop();
      }
      public function resume():void {
        play(lastPosition);
      }
      protected function updatePositionDisplay(event:Event):void {
        if (channel && sound) {
          positionTF.text = (channel.position/1000).toFixed(2) + " / " +
                            (sound.length/1000).toFixed();
        }
      }
      protected function onControlClicked(event:TextEvent):void {
        switch (event.text) {
          case "pause":
            (isPaused)? resume() : pause();
            isPaused = !isPaused;
            break;
          case "rw": rewind(); break;
          case "ff": fastForward(); break;
        }
      }
    }
  }
}
```

Tip

Make sure you stop the channel that's playing before playing again at a different position. If you don't stop the channel, you will hear a second instance of the sound playing on a different overlapping channel. ■

You might want to try adding onto this simple audio player as an exercise.

Applying Sound Transformations

Just as you can transform graphics to alter their appearance (as I show in Chapter 34, "Geometric and Color Transformations"), you can transform sounds to alter the way they are mixed. This is done by modifying the SoundTransform associated with a sound. Instances of the SoundTransform class define several properties, modifying which immediately alters how the associated sound is mixed.

The six Number properties of SoundTransform control the way a sound is mixed:

- volume — The volume, from 0 (completely muted) to 1 (maximum gain).
- pan — The balance of the sound between the left and right speaker, from −1 (play entirely out of the left speaker), through 0 (the source mix of the sound with no changes), to 1 (play entirely out of the right speaker).

- `leftToRight`, `leftToLeft`, `rightToRight`, `rightToLeft` — Mix the source channels to the output channels. Each contribution is measured as a number between 0 and 1. You can use these to modify the stereo mix between the left and right channels. Whereas panning "moves" the sound from the left to the right channel, these properties can modify both channels independently. You can use this, for example, to swap the left and right channels in a stereo audio track, a feat that panning alone cannot accomplish.

Simply retrieve the `SoundTransform` object associated with a playing sound (or the global sound mix) and modify these properties directly. Because changes are immediate, you can use ActionScript to animate these properties as well, producing smooth fades.

Working with a Sound's Metadata

If you subscribe to the proper events on the `Sound` class, you can be notified of a variety of useful information about a sound, such as how it's loading and properties of the sound itself.

Checking Load Progress

Similar to the `LoaderInfo` class, `Sound` objects have `bytesTotal` and `bytesLoaded` properties that expose the sound file's total size and the amount that is loaded so far. You can observe load progress in action by subscribing to the `ProgressEvent.PROGRESS` event. The `Sound` object also broadcasts `Event.OPEN` as the load begins, `Event.COMPLETE` when loading completes successfully, and `IOErrorEvent.IO_ERROR` if loading fails. See Chapter 27 to review these events.

Getting a Song's ID3 Data

When using MP3-encoded sound, you can retrieve even more information about the sound by accessing its ID3 data, if it's so included. ID3 tags add metadata to audio files, such as the song's name, artist, track number, album name, and so on. Any music you play with jukebox software such as iTunes, Windows Media Player, and the like is organized and sorted by this data. Of course, you can't be certain that a given song has some or any ID3 data, because all ID3 tags are completely optional.

You can access ID3 metadata of an MP3 through the `Sound`'s `id3` property. ID3 data is stored toward the beginning of an MP3, so it may be available before the song is completely loaded. If you're loading an MP3 over the network, listen for the `Event.ID3` event, which is broadcast as soon as ID3 information becomes available. Before this event is broadcast, all these tags are `null`.

Once available, the `id3` property is an object of type `ID3Info`. ID3 tags are key-value pairs defined by the ID3 specification. Flash Player 9 and later support the ID3 2.4 spec for song metadata. All ID3 keys have four-letter case-insensitive codes like `TALB` (album title) and `TCON` (genre). The more common of these tags are available through more readable properties of `ID3Info` such as `album` and `genre`. Regardless of this convenience, all properties including these are available as their capitalized four-letter property name under the `ID3Info` object. (`ID3Info` is a dynamic class, so it supports any property name.) In Example 31-4 you retrieve and display information about "Winter" from the MP3 file.

Note
Supported `ID3Info` shortcuts are `album`, `artist`, `comment`, `genre`, `songName`, `track`, and `year`. Find a full listing of all ID3 tags in the specs on http://id3.org/. ∎

EXAMPLE 31-4 `http://actionscriptbible.com/ch31/ex4`

Accessing ID3 Tags

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.media.*;
  import flash.net.URLRequest;
  import flash.text.*;

  public class ch31ex4 extends Sprite {
    protected var sound:Sound;
    protected var channel:SoundChannel;
    protected var songInfoTF:TextField;

    public function ch31ex4() {
      createChildren();
      var songurl:String = "http://actionscriptbible.com/files/winter.mp3";
      sound = new Sound(new URLRequest(songurl),
                        new SoundLoaderContext(1000, true));
      sound.addEventListener(Event.ID3, onMetadata);
      channel = sound.play();
    }
    protected function createChildren():void {
      var txtFormat:TextFormat = new TextFormat("_typewriter", 14, 0);
      songInfoTF = new TextField();
      songInfoTF.defaultTextFormat = txtFormat;
      songInfoTF.autoSize = TextFieldAutoSize.LEFT;
      addChild(songInfoTF);
    }
    protected function onMetadata(event:Event):void {
      songInfoTF.text = "NOW PLAYING:\n" +
                        sound.id3.songName + "\n" +
                        "from " + sound.id3.album + "\n" +
                        "by " + sound.id3.artist;
    }
  }
}
```

As an exercise, you might try adding the ID3 info from Example 31-4 to the audio player in Example 31-3.

Sampling Audio

Starting in this section, I'll dig into the more dynamic ways you can use sound in ActionScript. Playing audio is all good and well, but with some new improvements in Flash Player 9 and 10, the possibilities really open up. First, let's do some audio visualization.

The `SoundMixer` class mixes all the sounds Flash Player is playing into a single audio stream it sends to your speakers. You can query it for a snapshot or an analysis of the mixed-down audio it's currently playing. When you call the static method `SoundMixer.computeSpectrum()`, Flash Player samples the audio output at that moment, putting it in a `ByteArray`. Let's take a look at the method's signature:

```
function computeSpectrum(outputArray:ByteArray,
                         FFTMode:Boolean = false,
                         stretchFactor:int = 0):void
```

The first thing to note is the required `ByteArray` argument and `void` return type. To keep this method fast and efficient, it doesn't allocate a new `ByteArray` and return it every time you call it. Rather, you pass in an existing `ByteArray` and the method fills it up. You should reuse this `ByteArray` instance between calls to `computeSpectrum()` to avoid thrashing memory. The `outputArray` tightly packs 512 Numbers (that's 2KB or 2,048 bytes at 8 bytes per `Number`). Use `readFloat()` to get the `Numbers` out of the `ByteArray`, as you learned about in Chapter 13, "Binary Data and Byte Arrays." The data it returns can represent two kinds of data, as described in the next paragraph, but it is always stored as 512 values between −1.0 and 1.0, the first 256 of which represent values in the left channel and the second 256 values from the right channel.

Caution

When you sample audio, Flash Player not only mixes down all sounds playing in your application, but it may include sound produced by other Flash Player instances running, for example if you have two browser windows open, each playing its own audio.

Furthermore, when you request audio samples, *every* sound being played must be able to provide sample data. This means every sound needs to be accessible in the active security context. You can play sounds from any networked location, but to get low-level access including ID3 data and samples, a cross-domain file needs to approve the accessing domain.

Because there are no guarantees that the user isn't playing a sound in another Flash Player instance that you don't have access to, when you sample audio there's a real chance it may fail through no fault of yours. Take care to catch these errors. ∎

The `computeSpectrum()` method can give you the sample's waveform or its frequency spectrum, depending on the `FFTMode` parameter. By default (when `FFTMode` is `false`), the method samples the audio and gives you a waveform: the wave being output to your speakers by Flash Player during the sample time. For waveforms, the x-axis or index in the array represents time, and the y-axis or value at each index represents amplitude. For example, a low humming would be visible as a long sine wave. The distance between the peaks of the waves is the tone's frequency, and the height of those waves is its amplitude.

When you pass `true` as the `FFTMode`, Flash Player passes the waveform through a Fourier transform. (The `FFT` in `FFTMode` stands for Fast Fourier Transform.) This transform calculates the frequencies that contribute to the overall sound. So when you use `FFTMode`, instead of a waveform, you get a frequency distribution. The x-axis or index in the array represents different frequencies, from low to high, and the y-axis or value at each index represents the contribution of that frequency during the duration of the sample. For example, if you watch a frequency spectrum as a song while a well-defined drumbeat plays, you see repeated peaks shoot up near the low end of the graph in time with the drums. The x-position of the peak is the mean frequency of the drum.
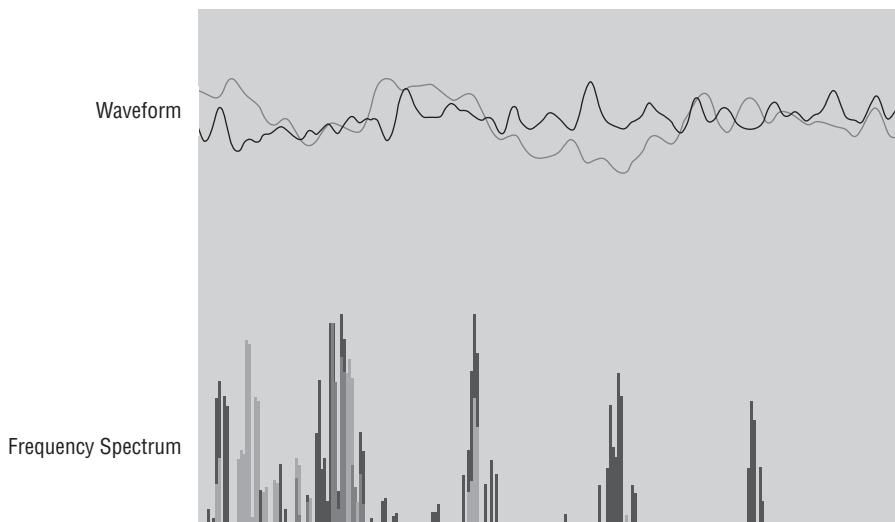
Both of these kinds of audio samples use a slice of time. You can also control how wide of a slice is used by setting the sample length. This is done implicitly with the `stretchFactor` parameter.

By default, and when the `stretchFactor` is 0, audio is sampled at 44.1 kHz. This means that the waveform you get back is 1/44100 of a second in duration (or the spectrum represents frequencies that occurred in that duration). When you increase the `stretchFactor`, it decreases the sample rate by that factor plus one, which stretches out the slice by that factor plus one. Let's say you use a `stretchFactor` of 1. This causes Flash Player to sample audio two times slower, at 22.05kHz, meaning that the samples are 1/22050 of a second, twice as wide. So the samples have effectively been stretched by a factor of two. You don't need to be so super-accurate with the sample rate when visualizing sound, because your ears are much more attuned to that kind of speed than your eyes.

You've probably seen both kinds of audio visualizations — waveforms and frequency spectrums — in action. They are compared in Figure 31-3. If you simply draw the output of `calculateSpectrum()` directly, you can get two passable kinds of audio visualizations, as shown in Example 31-5. But, of course, you can tie the produced data to all kinds of properties or put it through further calculations to make all kinds of interesting sound-reactive applications.

FIGURE 31-3

An example of both waveforms and frequency spectrums

EXAMPLE 31-5   http://actionscriptbible.com/ch31/ex5

Audio Visualization

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.media.*;
  import flash.net.URLRequest;
  import flash.utils.ByteArray;
```

EXAMPLE 31-5 *(continued)*

```
[SWF(frameRate="60",backgroundColor="#808080")]
public class ch31ex5 extends Sprite {
  protected var sound:Sound;
  protected var waveform:ByteArray;
  public function ch31ex5() {
    var songurl:String = "http://actionscriptbible.com/files/winter.mp3";
    sound = new Sound(new URLRequest(songurl),
                      new SoundLoaderContext(1000, true));
    sound.play();
    waveform = new ByteArray();
    this.addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  protected function onEnterFrame(event:Event):void {
    graphics.clear();
    SoundMixer.computeSpectrum(waveform, false, 2);
    //draw the left channel in blue
    graphics.lineStyle(0, 0x0000ff);
    drawChannel();
    //and the right in red
    graphics.lineStyle(0, 0xff0000);
    drawChannel();
  }
  protected function drawChannel():void {
    var W:Number = stage.stageWidth;
    var H:Number = stage.stageHeight;
    var SAMPLES:Number = 256;
    var xstep:Number = W/SAMPLES;

    graphics.moveTo(0, H/2);
    //assumes that the ByteArray is already at the correct location
    for (var i:int = 0, x:Number = 0; i < SAMPLES; i++, x+=xstep) {
      var amplitude:Number = waveform.readFloat();
      var y:Number = H/2 + (amplitude * H/2); //amplitude is from -1 to 1
      graphics.lineTo(x, y);
    }
  }
}
}
```

This example shows a basic and somewhat boring waveform visualization. There are hundreds of examples of exciting ones that you can find by searching for computeSpectrum on the internet or http://wonderfl.net/. See also the results of the visualization contest that took place on http://theflashblog.com/?p=197.

Extracting Audio

When you sample audio with `SoundMixer.computeSpectrum()`, you get a tiny slice of the complete audio mix at the current time. But in Flash Player 10 and above, you can do better: you can extract any portion of any single sound. This has many applications. You can use it to draw a preview of a whole sound clip for an audio editing application. You can use beat detection to try to match the tempo of a song. Or you can take an input sound, modify its audio data, and play back the modified audio — there you have an effect filter!

Version

FP10. Extracting sound is only possible in Flash Player 10 and later. ■

You can extract audio from a `Sound` with its `extract()` method. In this case, you call it on the `Sound` you want to sample, unlike `SoundMixer.computeSpectrum()`. Its signature is

```
function extract(target:ByteArray,
                 length:Number,
                 startPosition:Number = -1):Number
```

This method, too, passes the sample data by reference. The important value is not the return value so much as the `ByteArray target`. Calling this method fills up `target` with the sample data. The data is provided in 44.1 kHz stereo, again as a series of `Number`s between −1.0 and 1.0.

The `length` parameter specifies how many of those samples you want, and will, of course, determine the size of `target`. The `startPosition` parameter works in two ways. If you always omit it, repeated calls to `extract()` advance through the source `Sound`, pulling the next available samples since the last call. This makes it convenient to progressively process a whole sound. (You might want to spread out this processing over some time so as not to hang up Flash Player.) If you specify a value for this parameter, the returned audio data draws from the source sound starting at `startPosition` and ending at `startPosition+length`, expressed as numbers of samples. For example, a `startPosition` of 44100 and `length` of 22050 extracts the raw audio data between 1.0 sec and 1.5 sec.

Synthesizing Audio

Few will argue that the most compelling audio feature added in Flash Player 10 is the ability to generate dynamic audio. Developers clamored for this feature, and we got it. So use it!

Version

FP10. Flash Player versions 10 and later can play real-time synthesized audio. ■

To generate audio, simply create an empty `Sound` and `play()` it. The `Sound` requests audio data be fed to it periodically. It does this by dispatching a `SampleDataEvent`. You simply fill that request by filling the event object with audio data. As long as you keep up with the `Sound`'s requests, the sound keeps playing. The sound completes playing when you stop filling up the buffer or when you call `stop()` on the associated `SoundChannel`. Essentially, the `Sound` becomes an input stream to which you provide raw data. It keeps playing what it has buffered and asking for more, like a bucket with a hole in the bottom.

First let's look at the event. You'll subscribe to `SampleDataEvent.SAMPLE_DATA`. When you receive this event, Flash Player is asking for more audio samples to play. Regardless of when you get this signal, the `SampleDataEvent` event object tells you where in the audio stream it needs more data. You need to fill up the buffer with a continuous sound, but you're doing it piecemeal, so you need to make sure that every chunk of audio you add to the buffer lines up with the last. The `SampleDataEvent` object tells you where the requested sample goes in the audio stream with its `position` property. The position in the stream is measured in number of samples at a sample rate of 44.1 kHz (so a request with a `position` of 88200 starts 2 seconds into the audio stream).

The other property of `SampleDataEvent` is data, an empty `ByteArray` it provides you to fill up with data. Flash Player wants you to write between 2,000 and 8,000 samples in the buffer for every request. The less you provide, the more likely it is that the buffer will run out between calls to fill it up if Flash Player becomes busy. Write in raw audio data as a wave with `Number` values between −1.0 and 1.0, at a sample rate of 44.1 kHz, alternating left and right channels. Let's look at a simple example in Example 31-6.

EXAMPLE 31-6 `http://actionscriptbible.com/ch31/ex6`

Synthesizing Simple Tones

```
package {
  import flash.display.Sprite;
  import flash.events.SampleDataEvent;
  import flash.media.Sound;

  public class ch31ex6 extends Sprite {
    public function ch31ex6() {
      var sound:Sound = new Sound();
      sound.addEventListener(SampleDataEvent.SAMPLE_DATA, onBufferSound);
      sound.play();
    }
    protected function onBufferSound(event:SampleDataEvent):void {
      //for every sample, from the requested position through 8k samples later
      for (var t:int = event.position; t < event.position + 8192; t++) {
        //to generate a wave with any amplitude (volume) and frequency (pitch)
        //amplitude * sin(frequency * time)

        //the sin of this has 1 peak every seconds--is a 1Hz wave
        var tNormalized:Number = (t / 44100) * (2 * Math.PI);

        //write middle-C (261Hz) at full (1.0) volume to the LEFT channel
        event.data.writeFloat(1.0 * Math.sin(tNormalized*261.626));
        //write G4 (392Hz) at low (0.2) volume to RIGHT channel
        event.data.writeFloat(0.2 * Math.sin(tNormalized*391.995));
      }
    }
  }
}
```

In this example, you generate two sine waves. Air vibrating in a perfect sine wave like this produces this perfect — although a bit boring — tone. By alternating writing values, you can write to the left and the right channel independently, so the two alternating calls to `writeFloat()` produce two stereo channels. To play a mono sound, just write the same value twice. You might also notice that a chunk of 8,000 samples actually has 16,000 floating point numbers in it.

If you want to see how synthesized sound can improve an application, you might want to check out Example 37-5. In this example, you'll use filters along with sound synthesis to implement a repeating memory game. Check it out at `http://actionscriptbible.com/ch37/ex5`. The sound code is almost identical to the simple sine generator in Example 31-6, but with a fade-out.

Have fun! With some simple waves — sine, square, sawtooth — some noise, and some envelopes and effects, you can create a full range of retro sound. There are plenty of books and articles on audio synthesis. This series that spans five years of articles is superb: `http://bit.ly/reid-synth-secrets`. Or visit André Michelle's lab to have your mind blown with what you can do with Flash Player audio: `http://lab.andre-michelle.com/tag/audio/`.

Detecting Audio Capabilities

Now that you've seen what's possible, let's take a short sanity check. It's not a great idea to create an application that relies on audio alone. There are those of us who are hard of hearing or deaf — and those of us who don't have audio support on our computers. For the former category of users, you should come up with a strategy to support them, for example, providing a dialog to enable visual cues where audio would otherwise be used. For the latter category, you can check whether the client's computer has audio support by using the `Capabilities` class. The static `Capabilities.hasAudio` and `Capabilities.hasStreamingAudio` properties of this class tell you if the client can support sound or streaming sound files. Here's a simple check:

```
if (Capabilities.hasAudio) {
  trace("Audio OK!");
} else {
  trace("Audio not available");
}
```

Note
If you attempt to play sounds on a computer that does not have audio support, the `Sound.play()` function returns `null` instead of a `SoundChannel` object. ■

Summary

- The sound system is made up of several classes that work together. The Flash Player event framework is used to communicate the status of sounds loading and playing.
- A `Sound` object contains the sound data and loads and streams sound files. `SoundChannel` is a handle on the audio output from a `Sound` object. The `SoundMixer` controls the overall output of all the sounds, and `SoundTransform` controls how sounds are mixed.
- Sounds can be embedded, loaded from local files, or streamed from external files over the internet.

- ID3 data contains detailed metadata about an MP3 song, such as artist, genre, and song name. Access this information through a sound's `id3` property.

- You can sample audio as a waveform or a frequency spectrum to enable audio visualizations.

- In Flash Player 10 and up, you can extract, modify, and generate audio as it is playing.

Playing Video

I n this chapter, I'll cover a huge topic: video. Flash Player is, at the moment, the leading way to watch video content on the web. Although many technologies are nipping at its heels, none has the reach that Flash Player does. With wide adoption of the Open Screen Project (`http://www.openscreenproject.org`), the Flash Platform is poised to continue delivering content on not just computers but a plethora of thin and mobile devices.

To do real justice to all the activities and topics surrounding video in Flash, I'd need another thick book. Not surprisingly, several such tomes exist. For an expertly detailed reference, I recommend *Video with Adobe Flash CS4 Professional Studio Techniques* by Robert Reinhardt (Adobe Press, 2009). There are other such books and reference materials available in print and online.

In this chapter, I'll survey how the Flash Platform deals with video: how you can create, encode, deliver, and play it. Most of these activities fall outside of Flash Player, but they are important aspects of video in the Flash universe. After this overview, you'll learn about how to handle video inside Flash Player using ActionScript 3.0, building a simple video player from scratch.

FEATURED CLASSES

`flash.net.NetConnection`

`flash.net.NetStream`

`flash.media.Video`

Video and the Flash Platform

In this section, I'll overview the topics surrounding video and the Flash platform, without diving into code. I'll cover what Flash Player is capable of, the many kinds of video, and the path that video content takes from production through playback.

Video Sources

You can work with different kinds of video in Flash Player. Each of these is handled in a significantly different way, so it's important to make these distinctions now.

Embedded

Using Flash Professional, you can embed video into a SWF. Inside Flash Professional, the video appears on the timeline. The video data is included in the SWF, which can make for large files. Embedded videos are easy to manage and add because they don't require loading and are neatly encapsulated. On the other hand, you don't have much control over how an embedded video plays, and of course you're stuck with the video you embedded — you can't choose to play a different video at runtime.

If you need to integrate complex, prerendered animated effects into a design, embedded videos are a good choice. You can preview them inside Flash Professional, and no code is required to set up and play them. You can also manipulate them visually to match up perfectly to the nonmoving elements of the design. Embedded video is frequently used in transitions of high-production-value web sites to exact a cinematic, lush visual experience.

Embedded videos are inappropriate, however, for the kind of video you sit back and watch. A sensible video player is able to show you video from any source, not just videos you've embedded. Besides, embedding video is one additional step of preparation that's just not necessary.

Flash Player looks at embedded videos as subclasses of `Video`; you can also choose to embed the video in a `MovieClip` instance.

Video Files and Streams

In this chapter and in your programming career, you're probably going to spend most of your time with video files and video streams. These are external resources that you tell Flash Player to retrieve. The actual video data might be contained in a file on a server or the user's system, or it might be generated on the fly by a live broadcast or a friend's webcam. Video files and streams may differ in specifics but are handled similarly. As far as classifying kinds of video, any video that you load dynamically can be lumped together under this banner. You may simply hear this category called "Flash video," because it encompasses all the ways Flash plays video from the internet.

All video sites that use Flash Player, like YouTube, Vimeo, and Hulu, play video files or video streams. If the goal of your application is to play prerecorded or live video content, you'll use this kind of video.

Video that's played dynamically is loaded or streamed with a `NetStream` object and attached to a `Video` instance with its `attachNetStream()` method. I'll cover this in depth in the section "Implementing a Video Player."

Webcam

Video from a camera attached to or integrated with the user's computer can be accessed, recorded, and streamed to a media server. Camera video is a distinct kind of video that is attached to the screen and handled differently. Recording audio and video is covered in Chapter 33, "Capturing Sound and Video."

To display video from the user's camera, use a `Camera` instance and attach it to a `Video` instance on stage with the `attachCamera()` method.

Codecs and Container Formats

Let's talk about the video data. Video files have to store a huge amount of information. A typical video has about 30 frames per second. That's 30 images every second. If you were to simply store these

images sequentially, the resulting file would be huge, perhaps over a hundred megabytes for just a minute of video. Just as image formats compress the image data rather than storing the color of every pixel, videos are compressed, taking advantage of similarities between frames as well as compressing the image data. Just as there are different compression formats for images (like JPG and PNG), there are a variety of compression formats that Flash Player supports. In audio and video especially, these are called *codecs* because they have to be able to both *co*mpress and *dec*ompress the video data. During the (continuing!) development of Flash Player, newer codecs are added as they prove their utility. This means different versions of Flash Player support different codecs.

A video contains not only a moving image but usually an audio track as well. So video files actually use two kinds of codec: one for video and one for audio. Flash Player supports a variety of audio codecs.

Finally, all this audio and video data must be wrapped up together and packaged in a file somehow. Different *container formats* are used to bundle multiple streams of data and metadata into a file. For a long time, the FLV file container was your only option in Flash Player. Now a more advanced container format, F4V, has taken precedence over FLV. The F4V container format is compatible with MPEG-4, and Flash Player can even load files that properly use the MPEG-4 Part 12 container format, including MP4, M4V, M4A, and some 3GP and MOV files. Whether Flash Player can access the streams in these files is another question.

To play a media file, then, Flash Player has to both understand the container format and have a proper codec for each stream, whether it be audio or video.

The following audio codecs are supported in Flash Player 10 and up:

- MP3 — A flexible, if aging, codec
- ADPCM — A simple and therefore fast codec
- Nellymoser — A proprietary single-channel codec for speech
- AAC — A more recent, more open standard for multichannel audio; already widely used
- Speex — An open, higher-quality speech codec

These video codecs are supported:

- Sorenson Spark — An aging proprietary codec
- On2 VP6 — A proprietary codec with higher performance
- Screen Video — A codec for screencasts, used by Adobe Connect
- Screen Video 2 — Its successor
- H.264, aka AVC or MPEG-4 Part 10 — A recent, open, high-quality standard; already widely used

A comparison of these formats, which Flash Player versions support which, and their strengths and weaknesses is available at `http://bit.ly/video-compression-and-encoding`, an excerpt from the aforementioned book *Video with Adobe Flash CS4 Professional Studio Techniques*.

The most important things to know are that the F4V and MPEG-4 container formats, H.264 video, and AAC audio, are only available in Flash Player 9.0.115 and later. If you are dealing with high-quality video, I recommend that you use this combination. You should also know that if you use an F4V container, you can't use any of the proprietary older codecs. You may use AAC or Speex for audio and H.264 for video. You can see the compatibility of codecs and file containers, as well as a useful guide for which audio and video codecs usually go together, at `http://bit.ly/video-codecs-and-pairings`.

This is a lot of information, especially for anyone new to video. You can always come back to reference this information. The takeaway should be this: video consists of a compressed video stream, and possibly a compressed audio stream, packed into a container format.

Metainformation

Flash video can carry metadata along with it: information about the video itself, as well as additional streams such as a timed text track for subtitles or cue points. You can use cue points to trigger Action-Script code when a video is playing, and you can use metadata about the video to fill in information on a video player. The kind of metadata that is available is based on the container format and, of course, whether the data happens to be included, because it is entirely optional.

Metadata

Standard video metadata includes parameters such as the pixel dimensions of the video, duration, data rates, video frame rate, and audio and video codec types. This data will be made available to the `onMetaData()` function of the `NetStream`'s delegate. I'll show this in action later in this chapter.

Additionally, F4V/MPEG-4 files can contain XMP metadata. XMP is a standard for storing metadata and is used in nearly every digital camera. Using XMP, you can add a huge amount of data to your video files. This data is available to ActionScript as an XML document to the `onXMPData()` function of the `NetStream`'s delegate. XMP is heavily namespaced, so don't forget about XML namespaces if you want to parse this data!

Cue Points

Cue points fire off events at predetermined points during a video's playback. You can couple these cue points with data in key-value pairs. Use cue points to create interactive videos and content-aware applications.

In FLV files, cue points trigger the `onCuePoint()` function of the delegate. In F4V files, the cue points are instead embedded in the XMP metadata, so you can parse them out and wait for the specified timecodes, but the callback won't be fired automatically.

Subtitles and Closed Captioning

Subtitles are made much easier with the adoption of MPEG-4 Timed Text tracks (MPEG-4 Part 17). Use F4V or another MPEG-4 container format, and you can easily include subtitles, closed captioning, or any other timed text application. (Karaoke, anyone?) Nothing in Flash Player draws the text for you; you still have to decide what to do with the text. However, the text is reported to you as the video plays through the delegate's `onTextData()` callback function.

Delivery Methods

Although the codecs you use can affect the quality of a video, choosing how to deliver the video can completely change the user's viewing experience. Delivery is the process of getting the video data to the user's computer or device.

Local, Progressive Download

The most straightforward way to deliver video is over HTTP. If you're developing an application that lives on the web, like a web site or banner, chances are that your SWF itself and all its assets are delivered over HTTP. When you have a directory structure on a site that includes images, sounds, external SWFs, and so on, and you use relative URLs to load external resources like

```
loader.load(new URLRequest("../images/header.png"));
```

even though no `http://` appears in the URL, the resources are loaded from the same site as the application.

You can load videos just as easily as resources by using relative or absolute URLs. If the app is running locally — for example, if you're testing it during development — video files play from the drive on your computer. If the URL points to a web server, the video files are requested and delivered over HTTP like any other resource. These two cases are handled seamlessly depending on their URL, just like any other dynamic loading situation in Flash Player.

Loading a video off of a web server is usually called *progressive download*. Flash Player can play a video as it is downloading. Because the file contains video data from start to finish, as you load the file, you get access to later positions in the video. By loading data into a buffer as it arrives and playing video out of the buffer as it needs to appear on-screen, Flash Player can handle the video loading and playing at different rates.

Even better, all modern web servers can let you seek around the video by serving up the file starting at a requested offset rather than the whole file. (If the web server doesn't support this through HTTP headers, there are server-side scripts that do the same thing with HTTP request variables and that you can use to proxy access to your videos.) If the user decides to seek to a position in the video that hasn't been loaded, you can make a request for the video starting at or around that offset and start playing that video immediately.

Delivering video files through HTTP has some limitations, however. Because you're hosting your video on a web server, those files are available like any other file on the server. You can't easily control access to these videos. It's a simple matter for anyone to download the video file and keep it to replay at any time. Unfortunately, this can be a nonstarter for some content providers. Besides content protection, web servers don't provide services for connecting peers' video sources, connecting you to a live video source, true streaming, automatic bandwidth detection and stream selection, or playlists. All these things are supported, however, by specialized video servers.

Streaming

Various specialized servers are available that provide more advanced video services. For your reference, these are some of the available servers:

- Flash Media Server (including its editions Flash Media Interactive Server, Flash Media Streaming Server, and Flash Media Development Server), the standard in server software from Adobe. The different editions carry different price tags, starting at free. See an overview at `http://adobe.com/products/flashmediaserver/compare/`.

- Red5, a popular, free, and open source server written in Java (making it very portable). Available at `http://red5.googlecode.com/`.

- ElectroServer, `http://electro-server.com/`.

- Wowza Media Server, `http://wowzamedia.com/`.

Keep in mind that you can only choose to use a media server if your host supports it or you have the access to install and support it yourself.

First and foremost, a media server enables you to deliver true streaming Flash video. Video is streamed when the server and the client keep a connection open, the server sending the client only the video it needs to play next. The client and server stay in touch throughout playback and can even adjust the quality of the video to adapt to changes in available bandwidth.

Video streams are the only way to share live video feeds such as live events and webcam input. When sharing video streams from a client's webcam, you can either use a media server to republish the stream to the connected peers or use RTMFP (available in Flash Player 10 and up) to stream directly between two clients, using a matching service only to connect the two clients. This kind of video is called *peer-to-peer*. Adobe provides a free matching server, called Stratus, at `http://labs.adobe.com/technologies/stratus/`.

Streaming video is more easily controlled than progressive download. Not only can media servers apply special rules to the clients when they try to play a stream, but media servers that support RTMPE or RTMPS can send *protected streams* that are encrypted during transport. Both protected and unprotected streaming video are more difficult to save locally than a simple web-hosted video file. Protected streams have added security, making it more difficult (although still not impossible) to capture the source video.

True streaming uses a variety of protocols, some of which I've already mentioned. Because this array of technology has grown quite large, it is summarized in Table 32-1. Keep in mind that progressive download uses only one protocol: HTTP.

Beyond the selection of protocol, you can configure your media server in all kinds of ways to limit access to content, provide alternate streams for different viewers, provide different quality streams for different bandwidths, set up playlists, and so on. And outside of the scope of video, media servers can use their connection for Flash Remoting (see more in Chapter 28, "Communicating with Remote Services") so that you can share video and data simultaneously.

Delivery Networks

As video has eclipsed all other forms of traffic on the internet in its sheer abundance, the importance of delivery has increased, and more people have come to rely on delivery networks. A content delivery network, or CDN, is a system of servers throughout the internet that provide better access to data than a single central server. This is done by redundancy of computers and replication of data for reliability, and connecting users to servers with less load and that are closer in terms of network topology (and, as it follows in most cases, geography) for higher responsiveness.

Although I just described the technical aspect of a CDN with respect to video, the business of a CDN is more interesting. Depending on what your aim is, getting video to users can be a simple task — encode a single video, upload it to your web server, and be done with it — or a serious process — like encoding, tagging, captioning, setting up viewing rules for, organizing, and delivering the back catalog of every episode for every show a major broadcaster owns. An industry has evolved to assist businesses in the more complex end of that spectrum. Offerings vary, but most providers attempt to simplify this job with a front end that organizes huge media libraries as well as automates encoding and publishing. In some cases, CDNs have been adding these services, and in other cases, new companies have sprung up offering them under the banner of an Online Video Platform, or OVP.

TABLE 32-1

Protocols Used to Stream Video to Flash Player

Protocol	Full Name	Comments
RTMP	Real Time Messaging Protocol	Adobe-created (application layer) protocol over TCP, supporting video and audio streaming as well as AMF-encoded data.
RTMPT	RTMP Tunneled	RTMP packets are encapsulated in HTTP packets and sent over HTTP, usually on port 80, to avoid being blocked by firewalls.
RTMPS	RTMP Secure	RTMP tunneled like RTMPT but sent over a secure HTTPS connection.
RTMPE	RTMP Encrypted	RTMP using end-to-end encryption; faster than HTTPS and without requiring certificate management. Encrypts communications channel but using keys based on publicly available data and "secret" constants, presenting only a moderate barrier to attack.
RTMPTE	RTMPE Tunneled	RTMP using end-to-end encryption like RTMPE but tunneled in HTTP packets over port 80.
RTMFP	Real Time Media Flow Protocol	Similar to RTMP but uses UDP instead of TCP; other small changes.

These service providers may offer different combinations of delivery methods for Flash video. They aren't a true third option — as far as ActionScript goes, you'll still choose to deliver video over either HTTP or a media server — but they are so prevalent that if you do any video work, you're likely to run into one.

Encoding

Flash video must usually be *encoded* — that is, compressed and saved into a file of the proper format. Before Flash Player 9.0.115 added support for F4V and MPEG-4-like container formats, and widely used open codecs H.264 and AAC, re-encoding video for use in the Flash Platform with its proprietary FLV container was basically a prerequisite. Even with this support, the source for many videos is a high-quality master, so some re-encoding to get the video file slim enough to fit through normal-sized internet tubes is still called for.

The way you encode your video has no impact on ActionScript code. However, it's an important activity in bringing video to the Flash platform.

Adobe Media Encoder

Flash Professional ships with the de facto application for encoding video, Adobe Media Encoder. The app lets you batch-encode multiple files, add metadata and cue points, and do basic cropping and trimming.

Exporting from Applications

Installing Flash Professional also installs the Flash video codecs to your system so that video and animation applications will typically be able to export directly to an FLV or F4V file. For example, entries for "F4V (H.264)" and "FLV" should appear in the Format list when rendering a comp in After Effects. If the video is being used only for Flash, this may be a good option to cut one encoding step out of your workflow.

Video Encoding Services

There are other pieces of software and online services that encode videos for you. These can be of some advantage if you need to encode many files, especially to multiple formats at once. Video encoding may often be provided by an OVP.

Playback

The last step in video's journey to your users' eyeballs is getting it on-screen. But playing back video can be a much more involved job than slapping some moving things on the screen. First, let's briefly learn how Flash Player optimizes the path from video data to the screen.

Video Acceleration

As Flash Player has been under development, many changes have been made to improve video performance and take advantage of modern graphics hardware. All versions of Flash Player (that support AS3) can display full-screen video.

Starting in Flash Player 9.0.115, Flash Player uses the GPU to scale up and draw full-screen video. This occurs whether or not your Flash Player content is embedded in the browser with the proper WMODE set. This hardware scaling is also available using the `fullScreenSourceRect` parameter of `Stage`.

Starting in Flash Player 10.1, Flash Player additionally offloads the *decoding* of H.264 video to the GPU, if it is capable. This drastically decreases the workload borne by the CPU.

Video Players

The Flash Player API includes the classes necessary to load a video and put it on the screen. But not much more. As with many of the deeper capabilities of the Flash platform, the Flash Player API exposes low-level tools, enabling you to build your own framework or use an existing one. In this chapter you'll build a simple one just as an exercise to familiarize yourself with the video classes. You'll see that there is no video player built into Flash Player; even a feature as simple as play/pause must be written in ActionScript 3.0.

That said, there are several capable video players out there that you can use and, in some cases, customize, to play back video content in your Flash application.

Perfectly capable players are provided with Adobe tools. Flash Professional comes with a `FLVPlayback` component that you can include and skin. You can use this component with the authoring tool as well as pure ActionScript 3.0 code. The Flex 3 Framework comes with the `VideoDisplay` control, and the Flex 4 Framework includes an updated Spark `VideoPlayer` component.

A popular drop-in solution is the JW FLV Player (`http://longtailvideo.com`). Another nice one is flowplayer (`http://flowplayer.org`). There are dozens more commercial and free players,

all with a variety of features, licenses, and prices. If your large project is integrating with an OVP, this provider will likely have a player for you to use and customize, such as Brightcove's Brightcove Player.

Finally, don't forget that some of the biggest sites that host video content may allow you to embed their player's SWF into your own to play content hosted on the site. For example, if you host video content on YouTube, you can benefit from using its familiar and capable video player (see `http://bit.ly/youtube-embedding`).

It's important to realize that *all* these video players work slightly differently, support unique features, provide for various levels of customization, and integrate in their own way with your code. Plus, you can always write your own. It's not always an easy choice to make. If you're writing an application for which video isn't the primary use, you can probably exhale now, pick something free and simple, and move on.

Open Source Media Framework

As anyone following internet trends in the past few years knows, the growth of online video has big content owners scrambling to track and monetize their content. The relatively recent explosion of technology surrounding the delivery, tracking, and monetization of Flash video has transformed the business of developing a fully featured video player from simply annoying to ponderously complex. To paraphrase Will Law, to write a fully featured modern video player, you now have to be an expert at all of these:

- User interfaces, UX and design
- Dynamic streaming
- Playlist parsing
- Streaming and progressive delivery
- Content Delivery Network integration and authentication
- Ad network integration
- Tracking provider integration
- Social networking features
- Quality of Service reporting

It's getting a bit out of hand. Furthermore, as you just saw, the ecosystem of competing video players is not only challenging to evaluate but difficult to integrate with different providers for the features mentioned here.

Thankfully, Adobe and Akamai are leading the way to provide you with one media player to rule them all. Together, they started the Open Source Media Framework project, a free and extensible Action-Script 3.0 media player framework and a set of plug-ins for it. Not surprisingly, the project aims to provide all the features just listed. I don't mean to sound evangelical, but as a developer I'm as glad as anyone to download a free, open source, well-tested, extensible, and feature-rich video player; and the more developers that adopt the player, the more time that can be pooled to make that one player better rather than writing different versions of the same thing.

You can read all about the project and download the player at `http://opensourcemediaframework.com/`. The OSMF code comes bundled with the latest Flex SDK and Flash Builder 4.

633

Implementing a Video Player

Now that I've told you never to write your own video player, let's write a video player! Whereas a novice can drop in a video player, an expert understands the underlying code.

Flash video uses three principal classes: `Video`, `NetConnection`, and `NetStream`, which we'll go over now.

Video

I've been primarily talking about Flash video in this chapter — as opposed to embedded and camera sources — but all types of video use the `Video` object. Its job is simply to display the video content. You use this class to get video in the display list, so naturally it extends `DisplayObject`. This simple fact — that video is a first-class display object — is one of the powerful features of Flash Player. You can do with a `Video` what you can do with any `DisplayObject`: filter it, mask it, draw it to a bitmap, scale, translate, and rotate it, and in Flash Player 10, transform it in 3D space and apply shaders to it.

`Video`'s constructor optionally takes an initial size (width and height as `int`s). If you can match this up exactly to the size of the video source, you can avoid scaling, for sharp video and less CPU usage. Once created, you can attach different video sources to a `Video` object with these methods:

- `attachCamera(camera:Camera)` — Start using the specified `Camera` as the video source.
- `attachNetStream(netStream:NetStream)` — Start using the specified `NetStream` as the video source.

Pass `null` to these methods to detach the current video source.

Because `Video` is just a display class, it can't control the video stream it's displaying. You can, however, use it to toggle some quality options. You can use these properties to improve video appearance:

- `smoothing` — When set to `true`, uses higher-quality scaling when the video is scaled, at the expense of some processor utilization. In Flash Player 9.0.115 and higher, when smoothing is on, video is mipmapped for especially nice-looking results when scaled down by powers of 2.
- `deblocking` — For lower bit-rate video, you can use a deblocking algorithm to post-process the video, smoothing out sharp "blocky" artifacts. This increases processor utilization. Different `int` values trigger different deblocking algorithms, which you can look up in the AS3LR. In almost all cases, you can leave this as the default, which automatically applies deblocking when needed.

NetConnection

Unlike the HTTP-centric world of `URLLoader` and `URLRequest` that you saw in Chapter 27 and throughout the book, video can be delivered over a variety of protocols, so it uses a different class, `NetConnection`. You might remember this class from its other purpose as the connection to a remoting server from Chapter 28. Remoting and streaming video go hand in hand. They use the same family of RTMP protocols, and media servers provide both remoting operations and video streaming. A `NetConnection` represents a persistent connection to a media server. Once you have established a connection, you can use the single `NetConnection` to open multiple simultaneous streams and send remoting calls, if the server supports it.

To establish a connection to a media server application, use the `NetConnection`'s `connect()` method with a `String` specifying the hostname or IP, protocol, and application name:

```
var nc:NetConnection = new NetConnection();
nc.connect("protocol://host[:port]/appname[/instanceName]");
```

The protocol can be any of those enumerated in Table 32-1. You might want to consult the Adobe Flash Media Server 3.5 Developer Guide at `http://bit.ly/fms-dev-guide` if you need to set up and connect to a Flash Media Server.

Flash video works over plain old HTTP as well; nevertheless, you still use a `NetConnection` when playing Flash video over HTTP. This keeps the interface the same, avoiding complication. When using progressive download video, just create a placeholder `NetConnection` and `connect()` it to `null` rather than a valid media server endpoint.

NetStream

Once you have a `NetConnection` established, you can open streams on it using instances of `NetStream`. A `NetStream` represents a single, one-directional audio or video stream. For example, you might open a connection to a multiuser chat application and talk to three peers by publishing your own stream and opening the three other streams active in the room. In this example, you have four active `NetStream`s on the same `NetConnection`.

The `NetStream` object, finally, is used to monitor and control a video stream and access its metadata. First, though, you have to get a `NetStream` from a `NetConnection`, play the proper resource, and connect it to the `Video` object to give it a place to display.

Every `NetStream` must be linked to a `NetConnection`, which is done in the `NetStream` constructor. For HTTP progressive download videos, use a dummy `NetConnection` as previously discussed.

```
var ns:NetStream = new NetStream(nc);
```

Version

In Flash Player 10 and later, the `NetStream` constructor has a second, optional argument that can be used to set up peer-to-peer connections. ■

Next, play the resource using the `play()` method. This method takes a variable argument list. When playing a video file on a web server, just provide the URL to the file here. When streaming a video file or a live stream, use these parameters:

```
function play(name:Object, start:Number = -2, len:Number = -1, reset:Object = 1)
```

- `name` — The stream's name or filename, possibly with a format prefix. For MP3 files, use `"mp3:<stream name>"`. For F4V and other MPEG-4 container format files, use `"mp4:<stream name or filename>"`. For FLV files, use the stream name; a prefix is not necessary.

- `start` — The number of seconds into the video that playback should begin. Negative numbers can signify priorities in stream selection.

- `len` — Set to a positive number to limit playback to a specific number of seconds. −1 means "play through to the end."

- `reset` — Controls whether the stream is played immediately (default) or added to a playlist.

You already learned how to connect the `NetStream` to a `Video`:

```
video.attachNetStream(ns);
```

Now that the video is playing, you can monitor and control its playback with `NetStream` methods:

- `pause()` — Pauses the video stream. Does nothing to a paused stream.
- `resume()` — Resumes a paused video stream. Does nothing to a playing stream.
- `togglePause()` — Switches between paused and playing states.
- `seek(offset:Number)` — Moves the playhead to the keyframe nearest `offset` seconds or generates a new keyframe at that offset dynamically if the media server supports it.
- `time` — A read-only `Number` accessor that represents the position of the playhead in seconds.
- `soundTransform` — A `SoundTransform` object that lets you modify the mix of the video's audio stream. See Chapter 31, "Playing and Generating Sound," to learn about `SoundTransform`.

You can also use these methods and parameters for quality control:

- `bufferTime` — A `Number` specifying how many seconds of video should be buffered before playback begins. Pausing increases the buffer in Flash Player 9.0.115 and up. See the AS3LR for more detail on buffering behavior.
- `bufferLength` — A read-only `Number` accessor that returns how many seconds of video are currently in the buffer.
- `info` — A `NetStreamInfo` object that reports detailed statistics on the audio and video streams of the video in Flash Player 10 and later.
- `play2(param:NetStreamPlayOptions)` — Like `play()` but can be used to switch streams during playback, for example, to switch bitrates midstream. The `NetStreamPlayOptions` instance includes information on the new stream and old stream. Use this method to implement dynamic streaming, only available in Flash Player 10 and later and Flash Media Server 3.5 and later.

Finally, you can respond to video events, including those that provide metadata. `NetStream` events come in two varieties, both a little different than your garden variety Flash Player event, so they require some special attention.

First are `NetStatusEvent` events, which report on a variety of stream conditions. Instead of subscribing to each event type individually, all `NetStatusEvents` use the same and only event type, `NetStatusEvent.NET_STATUS`. Actual information about the event is stored instead in the event object's `info` property, which in turn has two properties: `level` and `code`. The `level` categorizes the importance of the event into `status`, `warning`, or `error`, and the type of event is stored as a string in `code`. To react to `NetStatusEvents`, you can subscribe to the single event type and use a `switch` statement in the event handler:

```
ns.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);

function onNetStatus(event:NetStatusEvent):void {
  switch (event.info.code) {
    case "NetStream.Buffer.Empty":
      displayBufferingScreen();
      break;
```

```
        case "NetStream.Buffer.Full":
          hideBufferingScreen();
          break;
        case "NetStream.Play.Stop":
          enablePlayButton();
          break;
      }
    }
```

Normally, these would be different events.

The other class of video events doesn't even use `Events`, but rather calls functions that you define in a delegate object. Simply define these functions in a class or store them in an `Object`, and tell the `NetStream` to use that instance as its delegate by assigning it to the `client` property of the `NetStream`. When the video event occurs, `NetStream` looks for the appropriate function in its delegate object, calling it with any associated parameters. The `client` property defaults to `this`, so you could also simply subclass `NetStream`, adding these delegate methods to the class itself and letting the class be its own delegate. Delegates can define these functions:

- `onCuePoint(cuePointEvent:Object)` — Called when an FLV cue point is reached by the video playing. The `cuePointEvent` object contains the following:
    - `name:String` — The name of the cue point as encoded.
    - `parameters:Array` — An associative array of key-value pairs stored with the cue point.
    - `time:Number` — The time in seconds when the cue point was fired.
    - `type:String` — The type of the cue point: `navigation` or `event`.
- `onPlayStatus(netStatusInfo:Object)` — Called when a stream has finished playing or switches to another stream, passing an object with the same properties as the `info` property of a `NetStatusEvent`. For streaming video.
- `onMetaData(metadata:Object)` — Called when metadata is available. The passed `metadata` object includes parameters like the `duration` in seconds, the `framerate`, the `height` and `width` of the video, and so on.
- `onXMPData(xmpData:Array)` — Called when XMP metadata is available. See the section "Metainformation" for more details. You can access the raw XML in `xmpData["liveXML"]`. Available in Flash Player 10 and later.
- `onTextData(textData:Object)` — Called when an MPEG-4 Timed Text track in the video hits text data during playback. The `textData` object has a `trackid` property that stores the track number of the text track and a `text` property that stores the text itself. Available in Flash Player 9.0.115 and later.
- `onImageData(imageData:Object)` — Called when an image track in an MPEG-4 file has image data. The `imageData` object has a `trackid` that stores the track number of the image track, and a `data` property that stores the image data in a `ByteArray`. You can use `Loader.loadBytes()` to display the image. Available in Flash Player 9.0.115 and later.

That's a lot of stuff you can do with a `NetStream`! By using these features, you can build a nice video player that monitors quality and delivers a great experience for everyone.

Putting It All Together

Let's see how the pieces fit together by making a simple video player. Example 32-1 isn't something you should go and use in a production environment, but it demonstrates the major video playback features that Flash Player provides.

A Simple Video Player

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.text.*;
  import flash.utils.Timer;
  public class ch32ex1 extends Sprite {
    protected var video:Video;
    protected var nc:NetConnection;
    protected var ns:NetStream;
    protected var nsDuration:Number;
    protected var playPauseButton:TestButton;
    protected var progressTF:TextField;
    protected var progressTimer:Timer;
    public function ch32ex1() {
      video = new Video(224, 168);
      video.x = video.y = 10;
      addChild(video);

      nc = new NetConnection();
      nc.connect(null);

      ns = new NetStream(nc);
      ns.client = {
        onCuePoint: onCuePoint,
        onMetaData: onMetaData,
        onXMPData: onXMPData,
        onPlayStatus: onPlayStatus
      }
      ns.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);

      video.attachNetStream(ns);
      ns.play("http://actionscriptbible.com/files/fire.flv");

      playPauseButton = new TestButton(24, 24, "||");
      playPauseButton.x = 10; playPauseButton.y = 180;
      playPauseButton.addEventListener(MouseEvent.CLICK, onPlayPauseClicked);
      addChild(playPauseButton);
```

```
  var rwButton:TestButton = new TestButton(24, 24, "<<");
  rwButton.x = 39; rwButton.y = playPauseButton.y;
  rwButton.addEventListener(MouseEvent.CLICK, onBackClicked);
  addChild(rwButton);

  var fwButton:TestButton = new TestButton(24, 24, ">>");
  fwButton.x = 68; fwButton.y = playPauseButton.y;
  fwButton.addEventListener(MouseEvent.CLICK, onForwardClicked);
  addChild(fwButton);

  progressTF = new TextField();
  progressTF.height = 24; progressTF.width = 140;
  progressTF.x = 97; progressTF.y = playPauseButton.y + 4;
  progressTF.defaultTextFormat = new TextFormat("_sans", 11, 0,
    false, false, false, null, null, "center");
  progressTF.selectable = false;
  addChild(progressTF);

  progressTimer = new Timer(100);
  progressTimer.addEventListener(TimerEvent.TIMER, onUpdateProgress);
  progressTimer.start();

  var fsButton:TestButton = new TestButton(82, 24, "fullscreen");
  fsButton.x = playPauseButton.x; fsButton.y = playPauseButton.y + 29;
  fsButton.addEventListener(MouseEvent.CLICK, onFullScreenClicked);
  addChild(fsButton);
}
protected function onNetStatus(event:NetStatusEvent):void {
  switch (event.info.code) {
    case "NetStream.Play.Stop":
      ns.seek(0);
      ns.pause();
      playPauseButton.label.text = ">";
      break;
  }
}
protected function onPlayStatus(event:Object):void {
  trace(event);
}
protected function onCuePoint(cuePointEvent:Object):void {
  progressTF.text = "[cue point] " + cuePointEvent.name;
  progressTimer.delay = 750;
}
protected function onXMPData(data:Object):void {
}
protected function onMetaData(obj:Object):void {
  nsDuration = obj.duration;
}
protected function onUpdateProgress(event:TimerEvent = null):void {
  progressTF.text = ns.time.toFixed(1) + "s / "
                    + nsDuration.toFixed(1) + "s";
  progressTimer.delay = 100;
}
```

continued

EXAMPLE 32-1 *(continued)*

```
    protected function onBackClicked(event:MouseEvent):void {
      var seekTime:Number = ns.time - 1;
      seekTime = Math.max(0, seekTime);
      ns.seek(seekTime);
      onUpdateProgress();
    }
    protected function onForwardClicked(event:MouseEvent):void {
      var seekTime:Number = ns.time + 1;
      seekTime = Math.min(seekTime, nsDuration);
      ns.seek(seekTime);
      onUpdateProgress();
    }
    protected function onPlayPauseClicked(event:MouseEvent):void {
      if (playPauseButton.label.text == ">") {
        ns.resume();
        playPauseButton.label.text = "||";
      } else {
        ns.pause();
        playPauseButton.label.text = ">";
      }
    }
    protected function onFullScreenClicked(event:MouseEvent):void {
      stage.fullScreenSourceRect = video.getRect(stage);
      stage.displayState = StageDisplayState.FULL_SCREEN;
    }
  }
}
import flash.display.*;
import flash.text.*;
class TestButton extends Sprite {
  public var label:TextField;
  public function TestButton(w:Number, h:Number, labelText:String) {
    graphics.lineStyle(0.5, 0, 0, true);
    graphics.beginFill(0xa0a0a0);
    graphics.drawRoundRect(0, 0, w, h, 8);
    label = new TextField();
    addChild(label);
    label.defaultTextFormat = new TextFormat("_sans", 11, 0, true, false,
      false, null, null, "center");
    label.width = w;
    label.height = h;
    label.text = labelText;
    label.y = (h - label.textHeight)/2 - 2;
    buttonMode = true;
    mouseChildren = false;
  }
}
```

This is a basic, imperfect player that demonstrates some of the fundamentals of player technology:

- Loading a progressive download video
- Connecting `NetConnection`, `NetStream`, and `Video`
- Setting up a `NetStream` delegate and callbacks and listening to `NetStatus` events
- Watching for metadata and cue points
- Controlling and monitoring playback with `NetStream`
- Scaling full-screen video

Dealing with `Video`, `NetStream`, and `NetConnection` is the key to successfully integrating video into your Flash applications.

Summary

- The Flash Platform is a powerful vehicle for video online.
- Embedded videos, camera video, streaming video, and progressive download videos are handled differently.
- Flash Player supports various video container file formats, audio codecs, and video codecs in different combinations. Support varies by Flash Player version.
- There is a suite of streaming protocols and media servers that use these protocols to deliver streaming video to Flash Player clients; files on web servers and local disks may also be used.
- Modern video players take care of a huge set of features, especially those that attempt to monetize online video.
- There are dozens of video players available, but if a standard emerges, it will likely be the Open Source Media Framework.
- `Video` handles display of video content, `NetConnection` establishes a connection to a media server or acts as a dummy connection, and `NetStream` controls a single video stream. All three work together to play video.

CHAPTER

33

Capturing Sound and Video

Too often, Flash Player is viewed strictly as a way to *show* things to the user. But Flash Player can also be a tool that the user can communicate with. A computer's inputs aren't limited to just a mouse and keyboard: most computers have microphones or cameras attached or integrated. Action-Script can use these peripherals with the `Camera` and `Microphone` classes. Video and audio streams can be saved using Flash Media Server or a compatible product, or they can be analyzed and recorded locally.

FEATURED CLASSES

`flash.media.Camera`

`flash.media.Microphone`

Video Input Using a Camera

To capture video, you first have to retrieve the appropriate `Camera` object. The user's operating system is responsible for reporting compatible video devices to Flash Player; not just webcams, but digital video cameras, and maybe the occasional TV tuner card, may be used as a video source. I'll generically refer to all of these as a "camera" for simplicity's sake. Also, it's quite possible that the user has no compatible video device connected.

Retrieving a Camera Object

Each accessible video device on the user's system appears as an instance of `Camera`, so you don't create a `Camera` instance yourself, but retrieve the proper one.

To get a single `Camera` object, call the static method `Camera.getCamera()`. The method takes the name of the camera you'd like to connect to, optionally. Leave the parameter out, and it returns the default camera (as best as Flash Player can tell which is the default). If there are no available devices, it returns `null`. You can get an array of all the cameras attached to the system with the static accessor `Camera.names`, but in most cases you should use `getCamera()` with no arguments; this gives the user a chance to pick the right camera.

When ActionScript tries to access the camera returned by `getCamera()`, Flash Player displays a dialog box that lets the user choose whether to allow or deny access to the camera, as shown in Figure 33-1. Make sure your application window size is at least 215×138 pixels; this is the minimum size Flash Player requires to display the dialog box. If the user has chosen to remember the camera settings for the domain hosting the SWF, or if he's used the Flash Player Settings Manager to globally set camera access, the dialog box does not appear.

Allow Camera Access dialog box

When the user responds to this dialog box, the `Camera` instance fires a `StatusEvent.STATUS` event. The event object's `code` property indicates the user's response: `Camera.muted` if the user denied access; `Camera.unmuted` if he permitted it. You can also find out if the request was accepted after the fact, without listening for the event, by using the `muted` property.

Viewing Video from a Camera

In most cases, you'll want to show the video coming from the camera back to the user. Frequently, you'll show feedback on top of the video, for instance placing a 3D scene in a video feed of a physical space for augmented reality, or showing image analysis at work, or simply as a visual cue for the user that the application is recording.

`Camera` is not a display object. To get it to display on-stage, you need to create a `Video` object and associate the two. Then the `Video` — which is a `DisplayObject` — can be added to the stage. Even if you never show the video on-stage, you need to create a `Video` object to grab image data from the camera.

In Example 33-1, you'll retrieve a `Camera`, ensure that the user approved it, add it to a `Video` object, and display it on-stage. You won't be doing anything exciting with the video (yet), but you'll repeat this scaffolding code every time you need to use the camera.

EXAMPLE 33-1   http://actionscriptbible.com/ch33/ex1

Obtaining and Displaying a Camera

```
package {
  import flash.display.Sprite;
  import flash.events.StatusEvent;
  import flash.media.Camera;
```

```
import flash.media.Video;
import flash.system.Security;
import flash.system.SecurityPanel;
public class ch33ex1 extends Sprite {
  protected var camera:Camera;
  protected var video:Video;
  public function ch33ex1() {
    camera = Camera.getCamera();
    if (camera) {
      camera.addEventListener(StatusEvent.STATUS, onCameraStatus);
      if (camera.muted) {
        Security.showSettings(SecurityPanel.PRIVACY);
      } else {
        attach();
      }
    }
  }
  protected function attach():void {
    video = new Video(camera.width, camera.height);
    video.attachCamera(camera);
    addChildAt(video, 0);
  }
  protected function onCameraStatus(event:StatusEvent):void {
    switch (event.code) {
      case "Camera.Muted": removeChild(video); video = null; break;
      case "Camera.Unmuted": attach(); break;
    }
  }
}
}
```

Notice that you can force Flash Player to show the privacy dialog box, which lets the user allow or deny access to the camera and microphone, by calling `Security.showSettings()`. If you so desire, you can also prompt the user to choose a different camera source by showing the camera panel of the settings dialog box instead, with `Security.showSettings(SecurityPanel.CAMERA)`. Users can always access these security panels themselves by clicking Settings in Flash Player's context menu.

Now that you've seen the "right" way to do things, know that you can simply go ahead with attaching the camera, even if it's muted, and nothing will happen. However, you should at least check that `camera` exists to avoid dereferencing `null`, which causes a runtime error.

```
var camera:Camera = Camera.getCamera();
if (camera) {
  var video:Video = new Video(camera.width, camera.height);
  video.attachCamera(camera);
  addChild(video);
}
```

Go ahead and use this code instead if you don't care whether the user allows you to access his camera or not.

Tweaking Camera Settings

There are a few settings of the `Camera` object that you can modify to tweak how video is captured, compressed, and displayed. These preferences are set by a few methods, and they're accessed through read-only properties.

The `setMode()` method sets the most important properties of the camera: the width, height, and frame rate that it will attempt to capture video at. Its arguments are

- `width` — The width of each frame the camera should attempt to capture.
- `height` — The height of each frame the camera should attempt to capture.
- `fps` — The number of frames the camera should attempt to capture every second.
- `favorArea` — Whether the camera driver will use the exact values you pass at the expense of performance (`true`) or come back with the closest matching native capture mode (`false`). Optional, defaults to `true`.

The camera's capture area is of particular importance. You can display the image at other sizes than the capture size (by attaching it to a differently sized `Video`), and it will be scaled. You can usually speed up the capture speed by using the minimum acceptable size and scaling up to the desired output size afterward.

I suggest always passing `false` to `favorArea`. Using a native capture mode means less work for the camera driver and a smoother capture rate. When not using `favorArea`, the properties of the camera's capture frames may be different from what you request. You can always check their current value with the `width`, `height`, and `fps` properties.

Example 33-2 demonstrates using a scaled-down capture frame and allows the camera driver to pick a more efficient size if you've picked a difficult one. The actual frame size used to capture depends on your camera driver and the size of the stage.

EXAMPLE 33-2 http://actionscriptbible.com/ch33/ex2

Setting Video Capture Properties

```
package {
  import com.actionscriptbible.Example;
  import flash.media.Camera;
  import flash.media.Video;
  public class ch33ex2 extends Example {
    protected const S:Number = 2; //scale down factor
    public function ch33ex2() {
      var camera:Camera = Camera.getCamera();
      if (!camera) return;
      camera.setMode(stage.stageWidth/S, stage.stageHeight/S, 90, true);
      var video:Video = new Video(camera.width*S, camera.height*S);
      video.attachCamera(camera);
      video.smoothing = false;
      addChildAt(video, 0);
```

```
        trace("Used", camera.width, "x", camera.height, "@", camera.fps, "FPS");
        trace("Stage size was", stage.stageWidth, stage.stageHeight);
      }
    }
  }
```

The following methods apply to compression setting when sending video across the network to a Flash Media Server.

- `setQuality()` — Sets video compression parameters `bandwidth` (bytes per second the compressed stream can use) and `quality` (amount of compression used) between 1 and 100 (highest quality and least compression). Set either of these to 0 to indicate that the other argument should be given priority. If you set both arguments and Flash Player can't stay under the required bandwidth while using the specified quality level, frames will drop.

- `setKeyframeInterval()` — Sets the distance between key frames when encoding video feeds from this camera.

- `setLoopback()` — Determines if the local view of the camera feed is compressed or uses the raw data available from the camera. By default, Flash Player shows the loopback video — that is, uncompressed video. To give the user a preview of the compression settings being used, pass `true` to this method.

Detecting Camera Activity

Flash Player monitors the motion in a video feed, and reports when motion occurs, if you want to trigger recording or image processing in response to motion. This is entirely optional.

When the contents of the video frame are changing significantly, the `Camera` instance broadcasts a `ActivityEvent.ACTIVITY` event. The event object has a Boolean `activating` property that tells you whether the camera detected that activity started (`true`) or determined that there's no activity (`false`).

The camera also defines a method to set the amount of motion that triggers the camera's activity event: `Camera.setMotionLevel()`. The method takes two parameters that determine the level of motion required to trigger activity (between 0 and 100) and the duration of nonactivity (in milliseconds) required to switch back to "inactive."

At any time, you can find out how much activity the camera deems is happening by accessing the `Camera`'s `activityLevel` property. Camera activity is measured by an integer in the range 0 to 100.

Capturing and Analyzing Camera Data

Once the camera has been attached to a `Video` instance, you can get the current frame captured by the camera as a bitmap using `BitmapData` and the `draw()` method. This technique will be discussed in Chapter 36, "Programming Bitmap Graphics." There's nothing more you have to do with the `Camera` instance; `BitmapData` takes care of everything.

Sound Input Using a Microphone

Audio input is somewhat of an underutilized feature in Flash Player. Audio input can be used to enable hands-free interaction, a feature well suited to mobile devices. With Flash Player 10.1, you get full access to the sound data being recorded by the microphone, enabling everything from pitch shifting and real-time audio effects to voice recognition. Otherwise, you can have access to the overall audio level and, of course, you can send audio data to a Flash Media Server.

Retrieving a Microphone Object

Much like `Camera`, the system retains one `Microphone` object for each recognized audio input device. These audio inputs may be used for much more than simply listening to the user's voice. Connect with a `Microphone` object that represents a plugged-in instrument, and you can write music editing and mixing software, for instance. Again, I'll generically refer to a compatible audio input device as a "microphone."

Retrieving a `Microphone` object is just like `Camera`, so be prepared for some repetition. To retrieve a `Microphone` object, use the static method `Microphone.getMicrophone()`, which can accept an optional parameter used to identify a specific microphone. It's best to leave this off and let Flash Player pick the default, or let the user pick which one.

```
var mic:Microphone = Microphone.getMicrophone();
```

After the `Microphone` is initialized, it dispatches a `StatusEvent` where the `Microphone` reports whether or not the user will allow the SWF access to the microphone. If the value of the `code` property is `Microphone.Muted`, the user didn't allow the SWF access to the microphone. The value `Microphone.Unmuted` means that the user allowed the microphone access. You can also find this out after the fact by examining the `Microphone` instance's Boolean `muted` property.

You can manually present the user with Flash Player security panels relating to microphone access, with the `Security.showSettings()` static method. Pass this `SecurityPanel.PRIVACY` to show her the allow/deny interface (again). Pass it `SecurityPanel.MICROPHONE` to allow her to choose the correct audio input device.

Playing Back Microphone Input

You can play back microphone audio locally by using the microphone *loopback* mechanism. Loopback is a term originating in analog electronics in which you patch your own output back in, perhaps with a short loop of wire: in this case, playing back what was just heard. Anyone who's tried this knows it's almost always a bad idea because of the inevitable feedback loop, even with a good headset.

Nonetheless, enable and disable the loopback with the `setLoopBack()` method. Pass it `true` to play through the speakers, or `false` to turn it off. If you do use the loopback, be sure to turn on echo suppression, covered in the next section, "Tweaking microphone settings." You can also use the `Microphone`'s `soundTransform` property to set a `SoundTransform`, which affects how the loopback audio is played through the speakers. See Chapter 31, "Playing and Generating Sound," for information on `SoundTransform`.

Tweaking Microphone Settings

The microphone hardware and compression settings may be set up by using the `Microphone` interface. Unlike `Camera`, not all of its settings-related properties are read-only.

Sound Recording Settings

Change the way the microphone records audio with the following properties and method:

- `gain` — Boosts the recording volume above what the operating system provides. Uses a `Number` between 0 and 100, defaulting to 50. The user can modify the gain in the Microphone Settings Panel.

- `rate` — The sample size used when recording audio. Set this value as a `Number` with a value that represents a valid frequency rate in kHz, rounded down, and limited to what the hardware supports. For example, 44 represents 44.1 kHz. You can set it to 5, 8, 11, 22, or 44.

- `setUseEchoSuppresion()` — Call this method with `true` to enable echo suppression. An audio processing algorithm goes over the recorded audio in real time, attempting to reduce echoes that may be produced by using loopback audio. The user can also enable or disable echo suppression in the Microphone Settings Panel.

- `noiseSuppresionLevel` — Only available when using the Speex codec (in Flash Player 10.1 and later), sets the maximum attenuation of noise in decibels. Defaults to −30 dB. The codec attempts to reduce noise below these levels.

The most important of these settings are probably the first two. If the user is too quiet or too loud, use `gain` to shift the volume into an acceptable range.

In Example 33-3, you display the microphone activity while changing gain with the keyboard. A "peak" light lets you know when the microphone is peaking, or louder than can be digitized. Try using the keyboard to move the gain around until there's no more peaking.

EXAMPLE 33-3 `http://actionscriptbible.com/ch33/ex3`

Adjusting Microphone Settings

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.media.*;
  import flash.system.*;
  import flash.text.*;
  import flash.ui.Keyboard;
  public class ch33ex3 extends Sprite {
    protected var mic:Microphone;
    protected var levelMeter:Shape;
    protected var peak:Shape;
    protected var gainTF:TextField;
    public function ch33ex3() {
      mic = Microphone.getMicrophone();
      //you can't see any activity without loopback being on
      mic.setLoopBack(true);
      //but loopback is annoying, so mute it!
      mic.soundTransform = new SoundTransform(0);

      gainTF = new TextField();
      gainTF.autoSize = TextFieldAutoSize.LEFT;
```

continued

EXAMPLE 33-3 *(continued)*

```
      gainTF.defaultTextFormat = new TextFormat("_typewriter", 12, 0);
      addChild(gainTF); gainTF.x = gainTF.y = 10;
      levelMeter = new Shape();
      levelMeter.graphics.beginFill(0x0000ff, 1);
      levelMeter.graphics.drawRect(0, 0, 80, -220);
      addChild(levelMeter); levelMeter.x = 10; levelMeter.y = 220 + 30;
      var s:Shape = new Shape();
      s.graphics.lineStyle(2, 0, 1, true, null, null, JointStyle.BEVEL);
      s.graphics.drawRect(0, 0, 80, 220);
      addChild(s); s.x = 10; s.y = 30;
      peak = new Shape();
      peak.graphics.beginFill(0xff0000, 1);
      peak.graphics.drawCircle(0, 0, 8);
      addChild(peak); peak.x = 120; peak.y = 30;

      addEventListener(Event.ENTER_FRAME, onEnterFrame);
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    }
    protected function onKeyDown(event:KeyboardEvent):void {
      switch (event.keyCode) {
        case Keyboard.ESCAPE:
        case Keyboard.SPACE:
            Security.showSettings(SecurityPanel.MICROPHONE); break;
        case Keyboard.UP: mic.gain += 1; break;
        case Keyboard.DOWN: mic.gain -= 1; break;
      }
    }
    protected function onEnterFrame(event:Event):void {
      levelMeter.scaleY = mic.activityLevel / 100;
      gainTF.text = mic.gain.toFixed(1);
      if (mic.activityLevel >= 100) {
        peak.alpha += 0.33;
      } else {
        peak.alpha *= 0.7;
      }
    }
  }
}
```

If you want to build on this example, try prompting the user to read a sentence aloud, and closely watch the `activityLevel`. You can adjust `gain` as he speaks to automatically calibrate the microphone. This is a much preferable solution than randomly picking a gain, and it's easier for the user than asking him to select it himself.

Sound Compression Settings

Several settings of `Microphone` let you tweak its compression settings, if its audio stream is being transferred to a Flash Media Server. These settings have no effect on the audio sample data you can analyze in Flash Player, which is always presented uncompressed.

Version

FP10. These properties are available only in Flash Player 10 and later. ∎

- `codec` — Sets the codec used to compress microphone audio. Options are enumerated as constants of `SoundCodec`. Currently Nellymoser and Speex are supported. Both are codecs tuned for speech compression. Speex is newer, lower latency, and provides more features. In Flash Player versions below 10, Nellymoser is available and used by default.

- `encodeQuality` — Applies only to the Speex codec. Controls compression quality, accepting an `int` between 0 and 10. Higher qualities take more bandwidth. (See the AS3LR for exact bit rates.)

- `framesPerPacket` — Applies only to the Speex codec. Sets the number of speech frames transmitted in each packet. Including more data in each packet increases latency but decreases required bandwidth.

Detecting Microphone Activity

You can detect microphone activity in several ways. Like the `Camera` interface, you can elect to be notified when significant audio activity starts and ends, and you can see the activity level (the overall volume) at any time. The Speex codec, available in Flash Player 10 and later, can even filter out nonvoice audio and notify you when the user is speaking.

Overall Activity

To determine the overall volume the microphone is currently detecting, use the `activityLevel` accessor. Complete silence is an activity value of 0, and peak volume registers a 100. Use the `gain` setting to make sure audio peaks don't get clipped over 100; Example 33-3 shows one approach.

You can also be notified of microphone activity by subscribing to the `ActivityEvent.ACTIVITY` event. The `activating` Boolean property of the event object tells you if activity is beginning (`true`) or ending. "Activity" refers to a duration of noise of a certain volume. For example, in creating a chat or broadcast application, when no one is talking, bandwidth can be saved because there is no need to send data. This information can also be used for visual feedback so that users know they (or others) are silent.

`Microphone` defines a read-only `silenceLevel` property that reports what the silence level of the microphone is. This can be set using `setSilenceLevel()`, which sets the minimum input level that should be considered activity and (optionally) the duration of silence necessary before activity is considered over. Silence values correspond directly to activity values. After gain is calibrated appropriately, any noise quieter than `silenceLevel` is not considered audio activity.

This method is similar in purpose to `Camera.setMotionLevel()`; both methods specify when the activity event is dispatched. However, these methods have a significantly different impact on publishing streams:

- `Camera.setMotionLevel()` is designed to detect motion and does not affect bandwidth usage. Even if a video stream does not detect motion, video is still sent.

- `Microphone.setSilenceLevel()` is designed to optimize bandwidth. When an audio stream is considered silent, no audio data is sent. Instead, a single message is sent, indicating that silence has started.

Voice Activity

The Speex codec can recognize characteristics of voice and use its own heuristics to determine activity. To use voice-based activity instead of levels-based activity, follow these steps:

1. Set the `codec` to `SoundCodec.SPEEX`.
2. Enable voice activity detection by setting `enableVAD` to `true`.
3. Set the silence level to 0 with `setSilenceLevel()`. This lets Speex do its job instead.

Version

FP10.1. Voice Activity Detection is available only in Flash Player 10.1 and later. ■

Capturing and Analyzing Microphone Data

With Flash Player 10.1 and later, you can get raw audio samples from the `Microphone`. Voice recognition is an obvious application; Didier Brun of `http://bytearray.org` was first out of the gate to develop such an application. See his original post at `http://bit.ly/bytearray-voice-gesture`.

Capturing audio from a `Microphone` is similar to dynamically sending audio to a `Sound` object. Simply subscribe to the `Microphone`'s `SampleDataEvent.SAMPLE_DATA` event. The audio data is stored as a sequence of samples, each with a floating-point amplitude between −1 and 1. If you want frequencies instead, you have to do the Fourier transform yourself — unless you want to mute all other sounds, set up a loopback, and use the `SoundMixer`'s `computeSpectrum()` method. See Chapter 31 for more information about sound data.

Example 33-4 draws the audio data to the screen in the simplest way possible.

EXAMPLE 33-4   `http://actionscriptbible.com/ch33/ex4`

Visualizing Captured Audio Data

```
package {
  import flash.display.Sprite;
  import flash.events.ActivityEvent;
  import flash.events.SampleDataEvent;
  import flash.media.Microphone;
  import flash.utils.ByteArray;
  public class ch33ex4 extends Sprite {
    protected var mic:Microphone;
    public function ch33ex4() {
      mic = Microphone.getMicrophone();
      mic.rate = 22;
      mic.addEventListener(SampleDataEvent.SAMPLE_DATA, onSampleData);
    }
```

```
    protected function onSampleData(event:SampleDataEvent):void {
      var waveform:ByteArray = event.data;
      var SAMPLES:Number = waveform.length / 4; //4 bytes per float
      var W:Number = stage.stageWidth;
      var H:Number = stage.stageHeight;
      var xstep:Number = W/SAMPLES;

      graphics.clear();
      graphics.lineStyle(0, 0);
      graphics.moveTo(0, H/2);
      for (var i:int = 0, x:Number = 0; i < SAMPLES; i++, x+=xstep) {
        var amplitude:Number = waveform.readFloat();
        var y:Number = H/2 + (amplitude * H/2); //amplitude is from -1 to 1
        graphics.lineTo(x, y);
      }
    }
  }
}
```

You can definitely get crazy with analyzing microphone input. You might even consider attempting to use one of the many C sound libraries using Alchemy (a product that runs C code in a virtualized C runtime inside the AVM2, allowing you to use C code alongside ActionScript 3.0).

Flash Media Servers

Cameras and microphones become even more powerful tools when they are paired with a media server. When the camera or microphone is paired with a media server, you can create video- and audio-enabled chat applications, video that can be saved on a server, and a wealth of other options. The canonical media server product is Adobe's own Flash Media Server, available in many editions and price points. Another option is Red5, an open source media server and general all-around Flash communication server. See Chapter 32, "Playing Video," for more details on alternative media servers.

Media servers allow you to stream data from the user's cameras and microphone to the server for recording or for relay to other users, as in a video chat application. Sending a video or audio stream to a media server is called *publishing* a stream. You can also easily *subscribe* to other video or audio streams, whether those come from a recorded source or another user as in a chat application.

To publish a video or audio stream, follow these general steps:

- Create a new `NetConnection` object.
- Establish a connection to the media server with the `NetConnection`'s `connect()` method.
- Create a new `NetStream`, associating it with the `NetConnection`.
- Attach a camera or microphone (or both) to the `NetStream`, using the `attachCamera()` and `attachAudio()` methods.

The same general approach applies to peer-to-peer connections.

Summary

- To retrieve a camera, call the static method `Camera.getCamera()`.
- To view the video feed from the camera, attach the camera to a `Video` instance with `attachCamera()`.
- Monitor the video activity by subscribing to the `ActivityEvent.ACTIVITY` event or watching the `activityLevel` property.
- Process video images by drawing the `Video` object into a `BitmapData`.
- To retrieve a microphone, call the static method `Microphone.getMicrophone()`.
- To listen to the audio from the microphone, call the `setLoopBack()` method.
- Monitor the audio activity by subscribing to the `ActivityEvent.ACTIVITY` event or watching the `activityLevel` property.
- Process microphone audio by listening to the `SampleDataEvent.SAMPLE_DATA` event.
- Publish video and audio to a media server with `NetStream`.

Part VIII

Graphics Programming and Animation

Geometric and Color Transformations

Geometric transformations, like moving, scaling, rotating, and skewing, and color transformations, like tint and brightness adjustment, can be represented as matrices. When you apply successive transformations, those transformation matrices are multiplied together so that the current state of an object can be represented by a single matrix, even though it may be the result of several transformations. Flash Player gives you access to these transformations with the `flash.geom.Transform` class. Using the properties of `Transform`, you can manipulate an object's geometric transformation matrices and its color transformations, giving you a more computer-graphics-centric interface to geometric transforms including position, scale, rotation, translation, and skew, and enabling basic manipulation of colors.

You can always manipulate the geometry of display objects with their properties like `x` and `rotation`. The transformation matrices introduced in this chapter provide a more powerful, mathematical interface to the same geometry. I hate to say it because it's so interesting, but you could consider this chapter optional.

To use transformation matrices, you'll need to understand some matrix math. In the section "Matrices and coordinates" I'll provide a brief introduction. When you get into 3D I'll provide another refresher in the section "Basic 3D concepts review." However, if you've never seen a matrix before, I highly recommend that you pick up a computer graphics or introductory linear algebra text while reading this chapter. I recommend some specific resources in the section "Basic 3D Concepts Review."

FEATURED CLASSES

`flash.geom.Transform`

`flash.geom`
`.ColorTransform`

`flash.geom.Matrix3D`

`flash.geom.Vector3D`

DisplayObject and the Transform Object

Every instance of a `DisplayObject` has a `transform` property, which always contains a `Transform` instance. The `Transform` object is a container for the different kinds of transformation matrices that can be applied to the display object. Its most important properties are those matrices:

- `colorTransform` — A `ColorTransform` object that represents a color transformation matrix

657

- `matrix` — The display object's current two-dimensional geometric transformation matrix
- `matrix3D` — The display object's current three-dimensional geometric transformation matrix

Only one of the two properties `matrix` and `matrix3D` may be set at a time, because the associated display object is definitively and exclusively either in two dimensions or three. If you set the display object's z value to something nonzero, a `Matrix3D` is created and placed in the `Transform`'s `matrix3D` property, and the `Transform`'s `matrix` property is set to `null`.

Version

FP10. The `matrix3D` property of `Transform`, the `Matrix3D` class, and the 3D-related properties (such as z) of `DisplayObject` are available in Flash Player 10 and later. ∎

Remember that the various transformation matrices of a display object are contained in its `Transform` instance. To access them, you need to dig a little deep:

```
var sprite:Sprite = new Sprite();
var colorTransform:ColorTransform = sprite.transform.colorTransform;
var transform:Matrix = sprite.transform.matrix;
var transform3D:Matrix3D = sprite.transform.matrix3D;
```

The transformation matrix `matrix` represents the transformation applied to the associated display object in its local coordinate space. For instance, if a display object's parent is rotated 45 degrees, its contents appear rotated at 45 degrees even if their local rotations are zero. Modifying the transformation matrix changes the properties of the display object in the same way that the properties such as x, y, `rotation`, `scaleX`, and so on do: with respect to its parent. But the `Transform` object also provides access to the concatenated transformation matrices. These matrices represent the orientation of the object in the global coordinate space (relative to the stage) and the combined color transform of the display object and all its ancestors. These matrices are available as the following properties of `Transform`:

- `concatenatedMatrix`
- `concatenatedColorTransform`

The concatenated matrices, however, are read-only. Because they combine transformations from potentially many display objects up the display tree, they are a derived value that wouldn't make sense to directly modify.

Now that you've seen how to find the transformation matrices, it's time to use them.

2D Affine Transformations

Affine transformations modify a coordinate space while preserving collinearity (straight lines don't become bent, parallel lines stay parallel) and ratios of distances (things can scale but only uniformly throughout the space). Perspective projections, for example, are not affine, because once-parallel lines converge on the vanishing point. When you manipulate a display object in 2D, the kinds of trans-

formations you apply to it are affine transformations. You should be more than familiar by now with these properties that transform the geometry of a `DisplayObject`:

- `x`
- `y`
- `rotation`
- `scaleX(width)`
- `scaleY(height)`

These properties change the orientation of the object and implicitly apply transformation matrices. By manipulating the transformation matrix, you can carry out the same kinds of transformations explicitly. These properties cover translation, rotation, and scaling. You can also construct transformation matrices that shear the object, a transformation that doesn't have an equivalent `DisplayObject` property.

Matrices and Coordinates

But why matrices? What does a matrix have to do with rotation, translation, or even position? Well, computers are awesome at math. They eat that matrix stuff up! So if you tell me to move something right by 2 and down by 3, I might do something ridiculous like this in my head:

```
x' = x + 2
y' = y + 3
```

Note
x′, read "x prime," indicates the "new x": the value of x after the transformation. ∎

How incredibly droll of me. Two equations. The computer suppresses a little laugh. It proceeds to school me: "You know, if you just got a little comfy with matrices, that could be one equation instead of two."

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} x \\ y \end{vmatrix} + \begin{vmatrix} 2 \\ 3 \end{vmatrix}$$

Sure, basically, it's the same so far. Instead of two values, x and y, you have a one-column matrix that contains those values. Matrices let you deal with whole systems of values at once. This is just a warm-up, though. To scale up by a factor of 2 horizontally and 3 vertically, I would use these equations:

```
x' = 2x
y' = 3y
```

You can't use matrix addition for this, obviously, and you can't simply multiply `[2 3]` by `[x y]` — the sizes are incompatible for matrix multiplication. But if you use a 2×2 matrix instead, you can do matrix multiplication:

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} \begin{vmatrix} x \\ y \end{vmatrix} = \begin{vmatrix} ax + by \\ cx + dy \end{vmatrix}$$

659

Or, in linear equations,

```
x' = ax + by
y' = cx + dy
```

To scale up `x` and `y`, you can ignore `y` in `x`'s equation and `x` in `y`'s equation. You can make their contributions nil by using 0 for the `b` and `c` terms in the matrix:

```
| x' |   | a 0 | | x |   | ax + 0y |   | ax |
| y' | = | 0 d | | y | = | 0x + dy | = | dy |
```

So if the computer wanted to scale up by a factor of 2 horizontally and 3 vertically, it might use the matrix

```
| 2 0 |
| 0 3 |
```

and multiply it by the coordinates:

```
| x' |   | 2 0 | | x |   | 2x |
| y' | = | 0 3 | | y | = | 3y |
```

This is a transformation matrix! In general, the form

```
| sx 0  |
| 0  sy |
```

is a transformation matrix that scales up by s_x horizontally and s_y vertically.

When a computer graphics algorithm is processing millions of points per second, it can do so with terrific speed by thinking of translations as matrices. The most twisted series of skews, scales, rotations, and translations is as simple as a single translation: they all boil down to multiplying a transformation matrix with the point's coordinate vector.

When we simple humans look at a rotation matrix, it's difficult for us to see a rotation. It seems pretty arbitrary compared to the `sprite.rotation = 90` statement you could otherwise use. But when a computer looks at a matrix, it sees a tight package of data that it can crank through a million multiplications in the blink of an eye. It may not look semantically meaningful, but the CPU cares not for such things — it needs to process numbers.

Using transformation matrices and the `DisplayObject` API, programmers have access to both perspectives on transformation.

Let's focus on the math again. You've derived (or guessed at) a 2×2 scale transformation matrix. The real transformation matrices Flash Player uses internally are 3×3 matrices. With this dimension — and a padding value in the coordinate vector to match — you can perform translations within the same matrix multiplication. Here's the multiplication decomposed:

```
| x' |   | a c tx | | x |   | ax + cy + tx |
| y' | = | b d ty | | y | = | bx + dy + ty |
| 1  |   | 0 0 1  | | 1 |   |            1 |
```

Or, in linear equations,

```
x' = ax + cy + tx
y' = bx + dy + ty
```

I've renamed a lot of the coefficients in this transformation matrix to match how the `Matrix` class names them. Can you see how the addition of 1 and `tx` and `ty` enables translation in the matrix multiplication? The translation elements of the matrix, `tx` and `ty`, are only ever multiplied by the new row 1, so they remain independent of x and y.

Kinds of Affine Transform and Their Matrices

With this general form for a 2D affine transform, this section examines the kinds of transforms mentioned, without necessarily deriving them.

The Identity Matrix

An *identity matrix* is a square matrix that, when multiplied by any compatible matrix, leaves it unchanged. Likewise, the identity matrix is a no-transform transform, but I thought it would be good to look at it anyway.

```
| x' |     | 1 0 0 | | x |     | x |
| y' |  =  | 0 1 0 | | y |  =  | y |
| 1  |     | 0 0 1 | | 1 |     | 1 |
```

As with all the examples, I encourage you to follow along and perform the matrix multiplication yourself. I promise it will help you gain insight into why the transformation matrices are constructed the way they are.

The identity matrix here leaves $x' = x$ and $y' = y$. Nothing goes anywhere.

Translation

Given the general form of the affine transformation matrix, you should be able to guess the form of a translation matrix.

```
| x' |     | 1 0 tx | | x |     | x + tx |
| y' |  =  | 0 1 ty | | y |  =  | y + ty |
| 1  |     | 0 0  1 | | 1 |     |      1 |
```

This scales x and y by a factor of 1 (that is to say, not at all), and then adds on the `tx` and `ty` factors as a translation.

Scale

You've already derived the scale transformation matrix in 2×2, so you should also have no trouble extending it to the 3×3 form.

```
| x' |     | a 0 0 | | x |     | ax |
| y' |  =  | 0 d 0 | | y |  =  | dy |
| 1  |     | 0 0 1 | | 1 |     | 1  |
```

Rotation

The rotation matrix looks a little odd to Cartesian-centric thinking:

```
| x' |     |  cos(θ)  sin(θ)  0 | | x |     | xcos(θ)+ysin(θ) |
| y' |  =  | -sin(θ)  cos(θ)  0 | | y |  =  | ycos(θ)-xsin(θ) |
| 1  |     |    0       0     1 | | 1 |     |               1 |
```

661

Note that although the `rotation` property expects degrees, this is not the natural unit for transformation matrices: θ must be measured in radians.

Skew

Skewing or shearing an object is simply a matter of shifting x proportional to y or vice versa.

```
| x' |     | 1 c 0 | | x |     | x + cy |
| y' |  =  | b 1 0 | | y |  =  | y + bx |
| 1  |     | 0 0 1 | | 1 |     |    1   |
```

To shear horizontally, set c to the amount of shear, and keep b at 0. To shear vertically, set b and keep c at 0. You can skew in both directions, too. It might be fun to play with how these values affect the shear, as shown in Example 34-1.

EXAMPLE 34-1 http://actionscriptbible.com/ch34/ex1

Skew Transforms

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.display.Stage;
  import flash.events.Event;
  import flash.geom.Matrix;
  public class ch34ex1 extends Sprite {
    protected var shape:Shape;
    public function ch34ex1() {
      shape = new Shape();
      shape.graphics.beginFill(0);
      shape.graphics.drawRect(0, 0, 300, 300);
      shape.graphics.endFill();
      addChild(shape);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      var s:Stage = stage;
      var xmouse:Number = (s.mouseX - s.stageWidth/2) / s.stageWidth;
      var ymouse:Number = (s.mouseY - s.stageHeight/2) / s.stageHeight;
      var m:Matrix = new Matrix(1, ymouse, xmouse, 1, 100, 100);
      shape.transform.matrix = m;
    }
  }
}
```

This example lets you see the effect that the b and c coefficients have on a transformation. It draws a rectangle that you can skew horizontally, vertically, or some combination by moving the mouse.

The Matrix Class

Flash Player provides a class for manipulating matrices, but unfortunately it is not designed for arbitrary-size matrices or even matrices with arbitrary content. It is specifically for use with affine transformation matrices. Using the `Matrix` class, you're restricted to the top two rows of a 3×3 matrix. This is all you'll need for transformation matrices, but it's no general-purpose matrix math library.

Example 34-1 already used a `Matrix` object and its constructor. The constructor takes from zero to six parameters: the coefficients a, b, c, d, `tx`, and `ty`, in that order. The matrix assumes these values, but you can also set them by their property name at any time after calling the constructor. The undefined third row always contains [0 0 1], as all 2D affine transformation matrices do. The default values for the arguments to the constructor construct an identity matrix when passed nothing.

Multiplying one transformation matrix by another combines their effects. You might visualize the series of transformations applied to a display object as a list; so adding another transformation to this series is sometimes called *concatenating* a transformation matrix. Of course, under the hood, Flash Player doesn't keep a list: it multiplies the successive transformation matrices. You can do this yourself with a `Matrix` by calling its `concat()` method. This method takes a `Matrix` as its only parameter and multiplies itself by the passed matrix. The result of the multiplication replaces the object's own value, and the transformation matrix represents the combination of the old transformation and the new. For example, if you move right by 5 and then subsequently move down by 10, the resulting transformation matrix should show a translation of (5, 10).

```
var m1:Matrix = new Matrix();
m1.tx = 5;
var m2:Matrix = new Matrix();
m2.ty = 10;
m1.concat(m2);
trace(m1); //(a=1, b=0, c=0, d=1, tx=5, ty=10)
```

Transform Methods

Rather than having to remember and reconstruct transformation matrices of the correct form, you can use convenience methods of `Matrix` to apply transformations. These transformations are immediately concatenated to (multiplied with) the contents of the `Matrix` instance, so their effects are cumulative.

The `Matrix` methods that correspond to affine transforms are

- `translate(dx:Number, dy:Number):void`
- `scale(sx:Number, sy:Number):void`
- `rotate(angle:Number):void`

These methods should be self-explanatory. The `angle` parameter to `rotate()` is measured in radians. You can also set a combination of transformations with one method:

```
function createBox(scaleX:Number, scaleY:Number,
                   rotation:Number = 0,
                   tx:Number = 0, ty:Number = 0):void
```

Note that, at minimum, you must include values for scale. The result of `createBox()` is equivalent to that of calling `identity()`, `rotate()`, `scale()`, then `translate()` with the appropriate parameters. The order that transformations are applied is important. More on order of applications later.

663

Utility Methods

The `Matrix` class has a few more methods used to reset the matrix, test its effects, and set a combination of transformations.

- `identity():void` — Resets the transformation matrix to an identity matrix
- `invert():void` — Performs the opposite of the transform in the transformation matrix
- `clone():Matrix` — Returns a new copy of the matrix
- `transformPoint(p:Point):Point` — Suitable for testing the transformation matrix; applies the transformation to a single point
- `deltaTransformPoint(p:Point):Point` — Suitable for testing the transformation matrix; applies the transformation, except for any translation it includes, to a single point

Order of Application

The order in which you apply transformations is critical. You can't reorder a series of transformations and expect the final orientation to be the same. A simple example consists of a translation and a rotation. If you translate an object right and then rotate it clockwise 45 degrees, you expect it to appear to the right of where it was, rotated. However, if you rotate the object and then move it to the right, after the rotation it has a new frame of reference, in which the translation doesn't go objectively to the right any more. In the new coordinate system, "right" has been rotated 45 degrees. Now "right" is diagonally down and to the right — to the objective frame of reference, that is!

In Example 34-2, two identical objects go through the same transformations in opposite order, pausing after every transform so that you can see how they diverge.

EXAMPLE 34-2 `http://actionscriptbible.com/ch34/ex2`

Order of Transformations

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.geom.Matrix;
  import flash.utils.Timer;
  public class ch34ex2 extends Sprite {
    protected var s1:Shape;
    protected var s2:Shape;
    protected var moves:Array;
    protected var moveNum:int;
    protected var timer:Timer;
    public function ch34ex2() {
      s1 = new Shape();
      s2 = new Shape();
      s1.graphics.beginFill(0xff0000, 0.5);
      s1.graphics.drawRect(20, 20, 100, 50);
      s1.graphics.endFill();
      s2.graphics.beginFill(0x0000ff, 0.5);
      s2.graphics.drawRect(20, 20, 100, 50);
      s2.graphics.endFill();
      addChild(s1);
```

```
      addChild(s2);

      var m:Matrix;
      moves = new Array();

      m = new Matrix();
      m.translate(200, 0);
      moves.push(m);

      m = new Matrix();
      m.scale(1, 3);
      moves.push(m);

      m = new Matrix();
      m.rotate(Math.PI/4);
      moves.push(m);


      moveNum = 0;
      timer = new Timer(1500);
      timer.addEventListener(TimerEvent.TIMER, nextMove);
      timer.start();
    }
    protected function nextMove(event:TimerEvent):void {
      var m:Matrix;
      var i:int;

      i = moveNum;
      m = s1.transform.matrix;
      m.concat(moves[i]);
      s1.transform.matrix = m;

      i = moves.length - moveNum - 1;
      m = s2.transform.matrix;
      m.concat(moves[i]);
      s2.transform.matrix = m;

      if (++moveNum >= moves.length) {
        timer.stop();
      }
    }
  }
}
```

The red rectangle goes through the transformations in the order they are created: translate, scale, rotate. The blue rectangle goes through the transformations in reverse order, and you can see them diverge quickly. Try modifying this code to create your own set of transformations.

Applying Transformation Matrices

When you retrieve the transformation matrix from the Transform object, a copy is returned. To apply a new transformation matrix, it's not sufficient to modify the transformation matrix inline: you only end up modifying a copy that is not in use. Instead, set a transformation matrix by assigning a

new `Matrix` to the `Transform`'s `matrix` property. Whether you assign a new matrix of your own creation or a modified copy of the preexisting matrix doesn't matter.

```
shape.transform.matrix.translate(10, 0); //doesn't move shape

var m:Matrix = shape.transform.matrix;
m.translate(10, 0);
shape.transform.matrix = m; //moves shape
```

Color Transforms

There are several ways to modify the color of display objects — using a`ColorTransform` or a `ColorMatrixFilter`. Using the `ColorTransform` object, you can quickly apply solid-color fills, tints, and brightness changes to display objects. For more complex color operations such as changing hue, saturation, and the like, use a `ColorMatrixFilter` object. The `ColorTransform` transformation matrix is part of the transformation API, described in this chapter, but the `ColorMatrixFilter` is part of the filter API, covered in Chapter 37, "Applying Filters." The `ColorMatrixFilter` uses matrix multiplication on a column matrix containing the color components R, G, and B, much like the transformation matrices in this chapter operate on column matrices containing position components x and y (and sometimes z). However, although it's also part of the transformation API, `ColorTransform` uses a simpler, nonmatrix method, which gives you less control than a `ColorMatrixFilter`.

Color Fills

The `ColorTransform` class defines a `color` property that lets you apply and retrieve a solid-color fill. The property applies a solid-color fill, not a tint. That means any contrast of color that is visible in the actual contents of a display object is indistinguishable after the fill is applied. For example, if you apply a solid-color fill to an object containing a rectangular photograph, the effect is a rectangle filled with a solid color. The contents of the photograph are no longer discernible.

The value of the `color` property is an unsigned integer color ranging from 0x000000 to 0xFFFFFF. This color is then applied to every pixel of the display object.

The following example constructs a new `ColorTransform` object, assigns a value of 0xFF0000 (red) to the `color` property, and then assigns the object to the `transform.colorTransform` property of `shape`:

```
var colorTransform:ColorTransform = new ColorTransform();
colorTransform.color = 0xFF0000;
shape.transform.colorTransform = colorTransform;
```

You can also retrieve the color that is currently applied to a display object using the `color` property. The property returns the current color applied to the `ColorTransform` object (if any):

```
var colorTransform:ColorTransform = circle.transform.colorTransform;
trace(colorTransform.color);
```

However, a `ColorTransform` object does not report the color of the artwork within a display object. It reports only on the color transform applied to the instance. For example, if a display object instance has a yellow square within it but no color transform applied to the instance, the

`transform.colorTransform.color` property does not return the number corresponding to yellow. It returns 0 because no color transformation has yet been applied.

Color Transformation Math

Color transformations using `ColorTransform` allow you to apply both a multiplier and an offset to each color channel, including the alpha channel. The transformation follows this formula:

```
r' = m_r r + o_r
g' = m_g g + o_g
b' = m_b b + o_b
a' = m_a a + o_a
```

Each color channel gets its own multiplier `m` and offset `o`. Use the `color` property of a `ColorTransform` to replace a display object's color wholesale. This shortcut sets all the multipliers to 0, and sets the offsets to the components of the destination color. The end effect is an entirely new color:

```
r' = 0r + o_r = o_r
g' = 0g + o_g = o_g
b' = 0b + o_b = o_b
```

Additionally, the alpha value remains unchanged. If the `ColorTransform` object contained an alpha scale or shift prior to setting color, it remains. Otherwise, the default scale and shift remain, affecting no change.

The `ColorTransform` class defines the following properties that correspond to the eight parameters in the preceding equation:

- `redMultiplier`
- `greenMultiplier`
- `blueMultiplier`
- `alphaMultiplier`
- `redOffset`
- `greenOffset`
- `blueOffset`
- `alphaOffset`

You can set the properties of a `ColorTransform` object using the constructor, or you can set them once the object has already been constructed. The constructor accepts from zero to eight parameters matching the properties in the preceding list, in the same order as they appear in the list. The following code constructs a new `ColorTransform` object with the default properties:

```
var colorTransform:ColorTransform = new ColorTransform();
```

Each of the multiplier properties (`redMultiplier`, `greenMultiplier`, and so on) can range from −1 to 1 and determines the percentage of the color component that is applied to each pixel. The multiplier properties default to 1, which means that each red, green, and blue component of each pixel's color is at 100 percent. By changing the multiplier properties, you can effectively apply a tint to a display object.

When the multiplier properties are changed, it doesn't remove the contrast and definition of the contents within a display object. For example, if a display object contains a photograph, applying changes

667

to the multiplier properties changes the tint of the photograph, but the subject of the photograph is likely still distinguishable. The following code applies a green tint to a `photo` by reducing the red and blue components to 0:

```
photo.transform.colorTransform = new ColorTransform(0, 1, 0, 1, 0, 0, 0, 0);
```

The offset properties (`redOffset`, `greenOffset`, and so on) add to and subtract from the red, green, blue, and alpha components of the color. The ranges for the offset properties are from −255 to 255. The default values are 0. Remember that when you store a color in a 32-bit integer, you use 8 bits for each channel, and each channel's total value can range from 0 through 255 (or 0xFF). You saw earlier how setting the `color` property uses the offsets to encode the new color, while eliminating the original color by setting its multipliers to 0.

The following applies a red tint to a photograph by setting the `redOffset` to 255:

```
var colorTransform:ColorTransform = new ColorTransform();
colorTransform.redOffset = 255;
photo.transform.colorTransform = colorTransform;
```

The multiplier and offset properties of a `ColorTransform` object can work in conjunction with one another. The multiplier properties affect the percentage of a color component already present within each pixel, whereas the offset properties add or subtract to or from the color component of the pixel. For example, if a pixel has a color of 0xFFFFFF (white), setting the `redMultiplier` to 0 or setting the `redOffset` to −255 has the same effect. However, if the color is 0x000000 (black), setting the `redMultiplier` property does not change the amount of red in the color, because anything times 0 is still 0. But you can set the `redOffset` to add red to the color.

Cross-Reference

How the three color components are packed into one `uint` is described in Chapter 13, "Binary Data and ByteArrays." ∎

Resetting and Combining Color Transforms

You can reset the colors applied to a display object by applying a `ColorTransform` object with the default properties as follows:

```
sprite.transform.colorTransform = new ColorTransform();
```

Just like a transformation matrix, you can also concatenate a new `ColorTransform` onto the existing one, combining their effects, using the `concat()` method.

```
var c1:ColorTransform = new ColorTransform(2); //multiply red
var c2:ColorTransform = new ColorTransform(1, 2); //multiply green
c1.concat(c2);
trace(c1); //(redMultiplier=2, greenMultiplier=2, blueMultiplier=1,...
```

Applying Color Transformations

In Example 34-3, you'll load an image into the player and then use a `ColorTransform` object to tint it based on the location of the mouse.

`http://actionscriptbible.com/ch34/ex3`

Color Transformations

```
package {
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.geom.ColorTransform;
  import flash.net.URLRequest;
  public class ch34ex3 extends Sprite {
    protected var l:Loader;
    protected var mode:int;
    public function ch34ex3() {
      l = new Loader();
      //photo   (CC-BY) Roger Braunstein
      //source  http://www.flickr.com/photos/rogerimp/3712472723/
      l.load(new URLRequest("http://actionscriptbible.com/files/caviar.jpg"));
      addChild(l);
      l.addEventListener(MouseEvent.CLICK, clickHandler);
      l.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveHandler);
    }
    protected function mouseMoveHandler(event:MouseEvent):void {
      var value:Number = l.mouseY / l.height;
      var colorTransform:ColorTransform = l.transform.colorTransform;
      switch (mode) {
        case 0: colorTransform.redMultiplier = value; break;
        case 1: colorTransform.greenMultiplier = value; break;
        case 2: colorTransform.blueMultiplier = value; break;
        case 3: colorTransform.alphaMultiplier = value; break;
      }
      l.transform.colorTransform = colorTransform;
    }
    protected function clickHandler(event:MouseEvent):void {
      if(++mode == 4) {
        mode = 0;
      }
    }
  }
}
```

When you test the application, you can move the mouse from top to bottom to adjust the red, green, blue, and alpha multipliers for the image. Each time you click the mouse, it advances to the next color component.

Retrieving the property from the `Transform` object returns a copy, so to update the display object with the new transformation, you must set a `ColorTransform` object to the `transform.colorTransform` property, rather than simply modifying it inline, just as with a transformation matrix. For more details, see "Applying Transformation Matrices," earlier in this chapter.

669

3D Transformations

In Flash Player 10 and later, `DisplayObjects` can be drawn in three dimensions. In Chapter 15, "Working in Three Dimensions," you learned how to position and orient 3D objects with `DisplayObject` properties. But just like their 2D brethren, 3D `DisplayObjects` can be transformed with transformation matrices. Using the transformation API in 3D is generally more powerful, and applicable, than in 2D. This is partially because the relative and absolute orientation of objects is increasingly hard to visualize and keep consistent in three dimensions, and partially because of the extended set of tools that the `flash.geom` package provides in Flash Player 10 and later.

Version

FP10. All topics covered in this section apply to Flash Player 10 and later. ∎

This section provides a refresher on some basic 3D math concepts that were only touched on in Chapter 15, making sure to include matrix math in the mix. This section covers not only affine transforms in 3D, but perspective projection transforms. With an overview of the theory, you can dive into the classes that aid 3D geometry in ActionScript 3.0, comparing them to similar classes that exist for 2D geometry.

Basic 3D Concepts Review

Before beginning, I want you to know that there are far better resources on 3D math than what you are about to read. I'm going to review some concepts at breakneck speed, mostly to introduce terminology and jog your memory. If you are new to 3D, there are many online and offline resources for learning the essential yet difficult concepts involved.

For beginners, I recommend *3D Math Primer for Graphics and Game Development* by Fletcher Dunn and Ian Parberry (`http://gamemath.com/`). It starts at square one and has an easy pace without skimping on explanation. It contains problem sets so you can test yourself. Best of all, it's mercifully short and mostly sticks to the real foundation material. Go get it as a companion if you are learning 3D concepts for the first time.

Online, the gamedev.net forums (`http://gamedev.net/`) are an endless treasure trove of information on graphics programming and beyond.
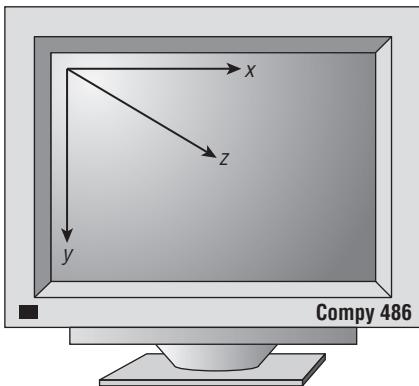
Now's a good time to review some of the terms and concepts needed to operate in 3D. For now it's important not to think about any ActionScript 3.0 classes you may have seen with similar names and just focus on the mathematical concepts. The class names and topics don't translate perfectly; the correct classes will be covered shortly.

Coordinate System

The coordinate system for 3D graphics in Flash Player 10 is a Cartesian space with three axes: x, y, and z. The origin is at the top-left corner of the screen. The +x axis proceeds toward the right of the screen, and the +y axis goes down the screen. The +z axis points away from you "into" the screen, and −z extends "out of" the screen, toward you, the viewer. The plane of the screen is where z = 0. The global coordinate space is pictured in Figure 34-1.

Global coordinate space superimposed on a screen
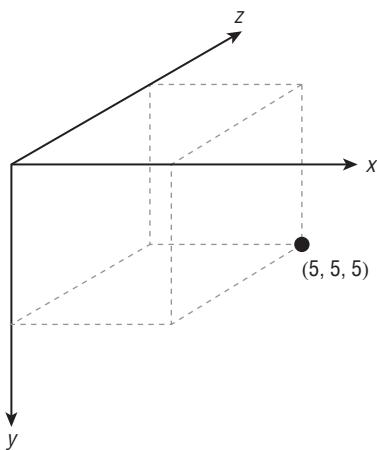
Points

In an *n*-dimensional Cartesian coordinate system, a location is described by *n* numbers. Each describes a component of the location along one of the space's axes. For instance, in two dimensions, a point is described by two numbers. You write this (x, y). The x value describes the point's position along the x-axis; the y value describes its position along the y axis. In three dimensions, you add a third axis, z, so you require a third variable to locate a point along the z-axis. A three-dimensional point has three values and is written (x, y, z). A real point, of course, uses real values in place of the variables, such as $(5, 5, 5)$, pictured in Figure 34-2. The dotted lines help you visualize the point's position in three dimensions.

A three-dimensional point
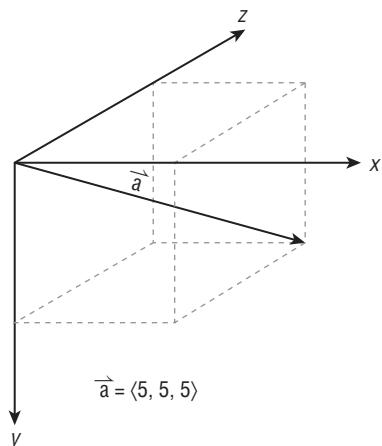
Points only have a location, no other properties. They represent a location in a coordinate space, but that coordinate space can be subjected to transformations. Your frame of reference is subjective. You've already seen this in two dimensions, as points in a child display object that is in motion are static within the display object's coordinate space, but they are in motion in the screen's coordinate space.

Vectors

A vector is similar to a point. (Remember, I'm not talking about the `Vector` class right now!) Vectors in *n* dimensions consist of *n* values. You can write a three-dimensional vector like this: **a** = <5, 5, 5>. The name of the vector, **a**, is written in bold or with an arrow atop it. This vector is pictured in Figure 34-3.

FIGURE 34-3

A three-dimensional vector **a**

$$\vec{a} = \langle 5, 5, 5 \rangle$$
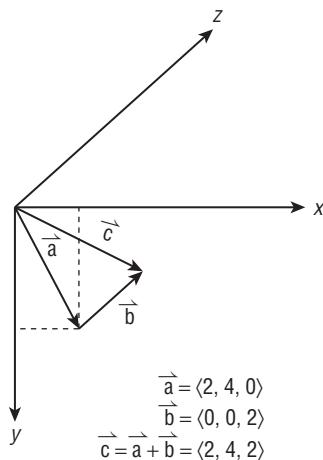
You'll notice that a vector in three dimensions is composed of three values, corresponding to the three axes, just like a point in three dimensions. But whereas a point represents a location in the coordinate system, a vector represents both a magnitude and a direction. You can see that the vector **a** points in one specific direction, and it has a beginning and an end, and thus a length or magnitude. The arrow indicates the direction of the vector. The beginning and end are not properties of the vector. They are merely by-products of the fact that this vector has been drawn starting at the origin, and if you follow the vector for its distance in its direction away from where you started, you inevitably arrive at its end. But the vector a never told you where to start, nor does any vector. I just picked the origin.

Vectors represent offsets or, if you like, displacements. The vector **a** doesn't tell you what to offset, where to start, or where it is. It tells you a direction and a distance. So it is less tied to the idea of an origin. But, as you saw, if you use a vector as a displacement from the origin, it perfectly represents a point.

Performing arithmetic on points doesn't make a whole lot of sense. Can you really "add" two locations? But performing arithmetic on vectors makes intuitive sense and is key to geometric transformations. The vector **a+b** has the same effect as following the vector **a**, then from there following vector **b**. This is pictured in Figure 34-4. Scaling a vector (multiplying it by a scalar) affects the magnitude of the vector but not its direction.

FIGURE 34-4

Summing two vectors

$$\vec{a} = \langle 2, 4, 0 \rangle$$
$$\vec{b} = \langle 0, 0, 2 \rangle$$
$$\vec{c} = \vec{a} + \vec{b} = \langle 2, 4, 2 \rangle$$

Vectors can be rotated, scaled, or even projected onto other vectors. You can compare them, measure their magnitudes and orientations, find angles between them, and even find a vector that's mutually perpendicular to two others. Because vectors and points are equivalent in terms of the information they carry, and because many of these useful operations apply well to vectors but not points, vectors are used far more often.

Matrices

Matrices have been introduced in this chapter. So, too, has the concept of a vector being written in matrix form. You can write a vector as a column matrix:

```
              | 1 |
a = <1, 2, 4> = | 2 |
              | 4 |
```

Writing a vector in matrix form doesn't change the vector or what it represents. It's not as if the vector is now a matrix, an incompatible kind of entity. On the contrary, a matrix is simply a way to organize a vector's information, and the change in forms is purely notational.

As you've seen with 2D transforms, matrix multiplication is an efficient replacement for multiple linear equations. In 3D, too, you can transform vectors through matrix multiplication:

```
| x' |   | 2 0 0 | | x |
| y' | = | 0 2 0 | | y |
| z' |   | 0 0 2 | | z |
```

Above, you uniformly scale up a vector <x, y, z> by a factor of 2 by using a 3×3 scale transformation matrix.

You should be familiar with the rules for matrix multiplication by now. Recall also that the identity matrix **I** is a square matrix with 1s down the diagonal. Following is the 3×3 identity matrix.

```
      | 1 0 0 |
I =   | 0 1 0 |
      | 0 0 1 |
```

You might notice that this is the same as scaling by a factor of 1, which should, of course, have no effect!

When using 3D transformation matrices, it's important that all the matrices are *invertible* — that is, for matrix $\mathbf{M}$ there exists a matrix $\mathbf{M}^{-1}$ such that $\mathbf{M}\ \mathbf{M}^{-1} = \mathbf{I}$. The matrix $\mathbf{M}^{-1}$ is the *inverse* of $\mathbf{M}$. You can check whether a matrix is invertible by calculating its determinant. Matrices with nonzero determinants are invertible. You won't have to know how to calculate a determinant to understand the concepts in this chapter; you can find out more in the recommended references. The inverse of a transformation matrix reverses the effects of that transformation. Unlike 3D transformation matrices, the 2D transformation matrices in ActionScript 3.0 are nonsquare, so they can't have inverses.

A final note on matrices and vectors. When you break up a matrix, its rows and columns are vectors, and it's not uncommon to look at the rows or columns that make up a matrix. For example, the second row (or the second column, as the matrix is diagonal) of the previous identity matrix is the vector <0, 1, 0>.

Orientation

Angular orientation can be measured in many ways. Orientation describes the rotation in three dimensions of an object. You can also talk about angular displacement instead of orientation, just as vectors are interchangeable with points. In any case, Flash Player gives you access to three ways of describing orientations:

- Euler angles — These define an orientation as a series of rotations around the coordinate space's axes. To achieve a certain direction, you rotate around the x-axis, then the y, then the z. You may be familiar with roll, pitch, and yaw. Those are Euler angles. Euler angles are famously easy to understand but infamously bad for interpolation and for gimbal lock, where two axes become coplanar during the rotation, eliminating a degree of freedom.

- Axis angles — These describe orientation as a scalar angular rotation around some vector, which is not necessarily an axis vector. The vector shows which direction the object is facing, and the rotation about it illustrates which way is facing "up." This form is also easy to understand and can simplify some calculations like movement in the direction of an object.

- Quaternions — These are notoriously weird representations of a rotation by a four-dimensional number, which is actually a kind of complex number in which there are three imaginary parts and one real part rather than one imaginary part and one real part. It can be written as the 4D vector <x, y, z, w>. Quaternions are great at interpolating orientations smoothly and are easy to concatenate.

3D Affine Transformations

Recall that for 2D affine transformations, an extra column was added onto the transformation matrix to enable translations (and extend linear transformations to affine transformations). Similarly, you can perform affine transformations in three dimensions by adding a fourth dimension to the matrix:

```
| x' |   | a b c t_x | | x |
| y' |   | d e f t_y | | y |
| z' | = | g h i t_z | | z |
| 1  |   | 0 0 0 1   | | 1 |
```

The 4×4 matrix on the left of the multiplication, of course, is the transformation matrix. (And now that you know the correct terminology, the 4×1 column matrices on the left and right are the post-transformation vector and the original vector.) The addition of the fourth column of the transformation matrix, and the dummy dimension of the vector spaces, allows for the t_x, t_y, and t_z coefficients to effect translation in the x, y, and z directions.

The scaling, translation, and even rotation matrices are of a familiar form to those in two dimensions. You should be able to guess the forms of the scale and translation matrices. Rotations around the three axes look like this:

```
          | 1     0      0   0 |
Rₓ(θ) = | 0   cos(θ) sin(θ) 0 |
          | 0  -sin(θ) cos(θ) 0 |
          | 0     0      0   1 |

          | cos(θ) 0 -sin(θ) 0 |
R_y(θ) = |    0   1     0   0 |
          | sin(θ) 0  cos(θ) 0 |
          |    0   0     0   1 |

          |  cos(θ) sin(θ) 0 0 |
R_z(θ) = | -sin(θ) cos(θ) 0 0 |
          |     0      0   1 0 |
          |     0      0   0 1 |
```

Now, because any angle can be represented as a sequence of rotations around the three axes, any angle can be represented as a product of these three matrices, each of which rotate about a single axis. This means that you can represent any orientation with the product of these translation matrices, which is in itself a 4×4 translation matrix. Likewise, any affine transformation can be composed into a single matrix with matrix multiplication.

Projection Transformations

ActionScript 3.0, however, actually provides access to the fourth dimension in the 4D vectors used with 4×4 transformation matrices, not to mention access to the bottom row of those matrices which has heretofore been filled with <0, 0, 0, 1>. The fourth dimension is called w, and it is not used as a real, independent dimension. Nor is it used in 3D space or in any transformation that takes place in 3D space (or rather, it is degenerate; the value 1 is always used for w). Where it is used, however, is during projection.

```
      | x |
      | y |
v  =  | z |
      | w |
```

Almost every screen today, like this piece of paper, is two-dimensional and displays a two-dimensional image. Yet it's easy to look at the page and discern 3D shapes. That's because these 3D shapes are drawn on a 2D surface in much the same way that your eyes focus an image on your retinas, or cameras focus an image on film or a sensor. Because your brain processes 2D projections of the 3D world all day long from the day you're born, it's adept at recognizing 3D shapes in artificial projections as well.

There are multiple ways you can project a 3D reality onto a 2D image. Artists have been doing this much longer than computers, so the art world is a much better place to look for different examples of
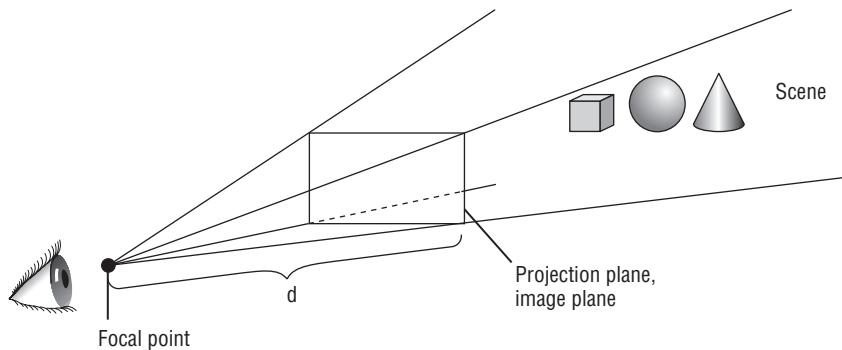
projections. Technical drawings and schematics are often drawn with an orthogonal projection, where there is no vanishing point: parallel lines stay parallel. Specifically, isometric projections are common, in which the axes are drawn at equal angles. In the Renaissance, perspective was finally cracked and popularized. When that looked too boring, cubists decided to break free from representational projections and draw impressions of a scene from fragmented viewpoints, where the connection between the point in 3D space and its position on the 2D image is more tenuous. To this day, some brave computer graphics researchers still toy with alternative methods of projection. But the clear winner in almost every produced image is, of course, the perspective projection.

Flash Player 10 lets you easily do perspective transformations. Of course, from the moment you use a display object in three dimensions, Flash Player is performing these transformations for you. So why do you care? One, to explain the use of w and the size of 4×4 transformation matrices. Two, to demystify how perspective projection works. And three, to be able to modify your projections.

At its core, perspective transformations are simple. To simulate foreshortening, where farther-away objects appear smaller, you simply scale down the coordinates of every point by some factor related to its distance from the projection plane. See Figure 34-5.

FIGURE 34-5

A perspective projection

In fact, you first calculate this factor in the w dimension. For a focal point at the origin and a projection plane at distance d from the focal point, you set the w of every point to the ratio of its distance from the focal plane:

$$
\begin{vmatrix} x' \\ y' \\ z' \\ w' \end{vmatrix}
=
\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{vmatrix}
\begin{vmatrix} x \\ y \\ z \\ w \end{vmatrix}
=
\begin{vmatrix} x \\ y \\ z \\ z/d \end{vmatrix}
$$

Then, to make this perspective factor affect the image, you convert 4D to 3D, scaling down all the coordinates by w. So the 3D version of this vector is <x/w, y/w, z/w>. In the end, you throw away z and draw the vector at <x/w, y/w>, where w is z/d.

For this trivial example, $x' = dx/z$ and $y' = dy/z$. Look closely: you're just scaling down the points depending on their distance. And that's what perspective is all about! With a more complex projection matrix, you can change the field of view and the position of the focal point.

Why does this all work? Because the four-dimensional space you work in is a *homogeneous* coordinate system. Its fourth dimension, w, extends all points in three space into a set of equivalent points in the

homogeneous coordinate system. For any vector <x, y, z> in real coordinates, there exists an infinite equivalent set of points of the form <kx, ky, kz, k> (for all values of k) in homogeneous coordinates. The homogeneous coordinate space of dimension n+1 still represents points of dimension n, but it "stretches out" each point in the n-dimensional space to a line of all possible points in dimension n+1 in which the homogeneous coordinate is a common divisor of the other coordinates. That's why, when you project from the 4D homogeneous dimensions into real 3D space, you divide x, y, and z by w. And it's also why you can treat vectors in three dimensions as existing in four dimensions where w = 1. (You can think of the "plane" at w = 1 as overlapping the "real" coordinate space.)

Another convenient property of homogeneous space is that it respects linear transformations. Its homogeneous properties are not broken when you apply a linear transformation in four dimensions. And, even better, affine transformations in 3D space are linear transformations in 4D space. When you translate in 3D space (using a 4×4 transformation matrix), you're actually *shearing 4D space*! By scaling just w with regard to distance from the image plane (the camera), you're scaling *all* the original coordinates because of the ratio relationship they have to the homogeneous coordinate.

If you find it hard to visualize, think of w as a redundant, invisible dimension that is used to control the scale of the other coordinates. At least half its purpose is simply enlarging vectors to 4×1 so that they can be multiplied by 4×4 transformation matrices with a fourth column that enables translation through matrix multiplication. The other half is, as you've seen, used in perspective projections.

3D Transformations in ActionScript

Up to this point, I've carefully avoided mentioning how ActionScript 3.0 utilizes these basic principles of 3D transformations. As the language has grown somewhat organically to support first 2D, then 3D geometry, there are some oddities that would have otherwise bogged down this introduction.

To get started, I'll compare the different classes in use for 2D and 3D math. All the classes in Table 34-1 are in the `flash.geom` package. You'll notice that the naming scheme is a bit inconsistent. `Point` becomes `Vector3D` instead of `Point3D`; `Point` is able to behave like a vector; the `Vector` class refers to a data structure and not a mathematical vector. Although the class `Vector` in the default package and the class `flash.geom.Vector` could easily coexist thanks to namespaces, the Flash Player architects have spared you this particular confusion.

TABLE 34-1

Geometry Classes in 2D and 3D

Concept	2D Class	Remarks	3D Class	Remarks
Point/Vector	`Point`	2 dimensions; has vector methods	`Vector3D`	4 dimensions
Transformation Matrix	`Matrix`	3×3, no access to third row	`Matrix3D`	4×4, access to all elements

There are more differences between the `Matrix3D` class and the `Matrix` class than just their dimension. The `Matrix3D` class, because it exists only in Flash Player 10 and later, uses other features that are also only available in Flash Player 10, notably `Vectors`. And the `Matrix3D` class provides some extremely helpful utilities to ease working in three dimensions. Even further utilities are included in the `Utils3D` class, which is covered in Chapter 40, "Advanced 3D."

677

Vector3D

In actuality, the `Vector3D` class is a four-dimensional vector of the form <x, y, z, w>. You can use Vector3D to represent vectors and points in 3D by ignoring the w coordinate, leaving it set to 1. The earlier discussion on homogeneous coordinate space explains the purpose of using a 4D vector to represent 3D space. You can use `Vector3D` all day without ever thinking about the fourth coordinate. Unless you dig really, really deep, Flash Player handles it for you.

This 4D vector can be used for purposes other than points and vectors in 3D space. It can also hold a row or column of a 4×4 matrix, represent a quaternion, or store the channels of a color as in <r, g, b, a>.

A `Vector3D` stores each coordinate in the properties x, y, z, and w. These properties can also be set at construction time. The constructor takes these parameters, all optionally:

```
function Vector3D(x:Number = 0, y:Number = 0, z:Number = 0, w:Number = 0)
```

You can leave w off when you're representing 3D points. Here you create a new vector for the point (5, 5, 5):

```
var v:Vector3D = new Vector3D(5, 5, 5);
```

The `Vector3D` class has some static constants that represent the three axes:

```
trace(Vector3D.X_AXIS); //Vector3D(1, 0, 0)
trace(Vector3D.Y_AXIS); //Vector3D(0, 1, 0)
trace(Vector3D.Z_AXIS); //Vector3D(0, 0, 1)
```

Recall that a vector encodes both a direction and a magnitude. You can easily access the magnitude of a vector through its `length` and `lengthSquared` properties, as Example 34-4 shows. These properties are read-only. In math notation, you write the magnitude of a vector **v** like |**v**|. These vertical bars, although I have typeset them the same, are not the same bars that are used to outline a matrix. In other words, |**v**| is not a 1×1 matrix but the notation for the magnitude of **v**.

EXAMPLE 34-4   `http://actionscriptbible.com/ch34/ex4`

Collected Snippets: 3D Geometry

```
var v:Vector3D = new Vector3D(8, 4, 1);
trace(v.length); //9
trace(v.lengthSquared); //81
```

Of course, the magnitude of a vector is its length, so you can calculate its magnitude with the Cartesian distance formula $\sqrt{x^2 + y^2 + z^2}$.

All the basic vector arithmetic you can do is readily performed with the `Vector3D` class.

- `add(a:Vector3D):Vector3D, subtract(a:Vector3D):Vector3D` — Performs vector addition and subtraction, returning the result of the sum. Vector addition sums each component individually:

  ```
  <x₁, y₁> + <x₂, y₂> = <x₁ + x₂, y₁ + y₂>
  ```

- `incrementBy(a:Vector3D):void, decrementBy(a:Vector3D):void` — Performs vector addition and subtraction destructively, replacing the contents of the vector with the result of the sum.

678

- `scaleBy(s:Number):void` — Performs scalar multiplication destructively, replacing the contents of the vector with the result of the scalar multiplication. Scalar multiplication multiplies each component of the vector with the scalar:

  `k <x, y> = <kx, ky>`

- `negate():void` — Negates a vector by multiplying it by −1. The operation is destructive, replacing the vector's contents with its negation.

- `equals(toCompare:Vector3D, allFour:Boolean = false):Boolean` — Compares two vectors, returning `true` if the vectors are equal. By default, this method compares the vectors as if they have three dimensions, ignoring any differences in `w`. By passing `true` to the `allFour` parameter, the method compares all four dimensions of the vectors. *Vector equality* is the equality of all the vectors' components:

  $<x_1, y_1> = <x_2, y_2>$ `iff` $x_1 = x_2$ `and` $y_1 = y_2$
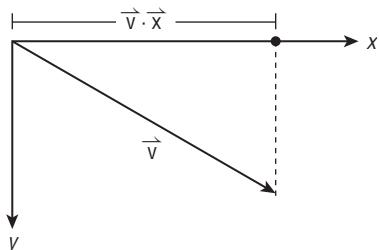
- `nearEquals(toCompare:Vector3D, tolerance:Number, allFour:Boolean = false):Boolean` — Compares two vectors, allowing for a certain amount of error. If the two vectors' components differ by less than the `tolerance` value, they are considered equal. In Chapter 7, "Numbers, Math, and Dates," you learned that floating-point numbers are not infinitely precise. Using this method, you can ignore small rounding errors and still determine if the vectors are meant to be equal, or you can provide a higher tolerance and determine if the vectors are "nearly equal." The `allFour` parameter acts as it does in the `equals()` method.

As I'm sure you noticed, some of the preceding operations are destructive and some are nondestructive. You can make copies of a `Vector3D` by calling its `clone()` method.

The `Vector3D` class also provides vector operations like dot product and cross product. The dot product of two vectors is useful for projecting one vector onto another vector. In Figure 34-6, vector v is projected onto the x-axis. The length of the projection is given by the vector's dot product with a unit vector in the x direction. In mathematical notation, dot products are written with a dot: **a** · **b**.

FIGURE 34-6

Using a dot product to determine a vector's projection onto an axis

The cross product produces (when possible) a vector that is mutually perpendicular to the two argument vectors. This operation is essential for computing the *normals* of polygons. A *normal vector* is a vector pointing "up" from a surface; with a flat surface like a polygon, this determines the direction
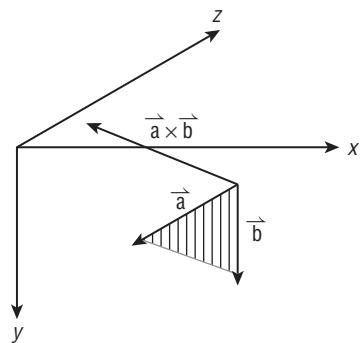
the surface is facing. Normals are used all over the place in computer graphics. Two of the simpler applications are for back-face culling and lighting, which you'll learn a little about in Chapter 40, "Advanced 3D." Figure 34-7 calculates the normal of a triangle by producing the cross product of two of its edge vectors. Cross products are noncommutative; the order of their arguments matter. They also depend on the handedness of the coordinate system. The 3D coordinate system in use in Flash Player is right-handed. This means if you curl the fingers of your *right* hand into a fist from the direction of the first vector to the second vector, your thumb will be pointing in the direction of the cross product. In math notation, cross products are written with a cross: **a** × **b**.

FIGURE 34-7

Using a cross product to determine a polygon's normal vector

These two operations are supported by the `dotProduct()` and `crossProduct()` methods:

```
var a:Vector3D, b:Vector3D, c:Vector3D;
a = new Vector3D(12, 47, 209);
trace(a.dotProduct(Vector3D.X_AXIS)); //12

a = new Vector3D(12, 0, 0); //points along +x
b = new Vector3D(0, 12, 0); //points along +y
c = a.crossProduct(b); //should point along +z, perpendicular to both
trace(c); //Vector3D(0, 0, 144)
```

Vector normalization is also frequently useful in computer graphics. Normalization keeps a vector's direction but loses its magnitude. This is done simply by scalar division by the vector's magnitude, so that the new magnitude is 1. Rather than take these two steps, however, `Vector3D` provides a `normalize()` method, which destructively normalizes the vector:

```
trace(c); //Vector3D(0, 0, 144)
c.normalize();
trace(c, c.length); //Vector3D(0, 0, 1) 1
```

Vector3D provides a few final useful operations:

- `Vector3D.angleBetween(a:Vector3D, b:Vector3D):Number` — Returns the angle between two vectors. If you think of the vectors as line segments starting at the origin, they form an angle on the plane between them. This static method measures that angle in radians. It doesn't care about the order of the vectors.

- `Vector3D.distance(pt1:Vector3D, pt2:Vector3D):Number` — Measures the distance between two points defined by vectors. This is identical to $|\textbf{pt2} - \textbf{pt1}|$, the magnitude of the vector between them.

- `project():void` — Destructively projects a vector from 4D homogeneous coordinates into 3D coordinates, by dividing x, y, and z by w.

Matrix3D

The `Matrix3D` class stores a 4×4 matrix. Although you can store any 4×4 matrix in it, it's designed for 3D transformation matrices.

Unlike the `Matrix` class, in which you could set the elements of the matrix directly through instance properties, the `Matrix3D` class has too many elements (16) for this to be very manageable. Instead, you can access its contents directly through a `Vector` of 16 Numbers (`Vector.<Number>`). Just like 2D transformation matrices, however, it's much easier to use the convenience methods of `Matrix3D` to construct and concatenate the proper matrices for you.

You can pass these 16 values to `Matrix3D`'s constructor or set them with the `rawData` property. The values are in column-major order — that is, they are stored from top to bottom and then left to right. If you don't pass a `Vector` of values to `Matrix3D`'s constructor, a 4×4 identity matrix is created. The `rawData` property returns a copy of the matrix's elements; to change them you have to assign a whole new `Vector` to the `rawData` property. (You can't modify it inline.) The default `Matrix3D` constructed when you pass nothing to the constructor is a 4×4 identity matrix. Of course, you can restore the matrix to the identity matrix at any time by calling `identity()` on it. All this is shown in Example 34-5.

EXAMPLE 34-5 `http://actionscriptbible.com/ch34/ex5`

Using Matrix3D

```
package {
  import com.actionscriptbible.Example;
  import flash.geom.Matrix3D;

  public class ch34ex5 extends Example {
    public function ch34ex5() {
      //create a default (identity) matrix
      var m:Matrix3D = new Matrix3D();
      prettyPrintMatrix(m); //identity matrix

      var v:Vector.<Number> = m.rawData;
      v[4] = 9.2;
      m.rawData = v;
```

continued

EXAMPLE 34-5 *(continued)*

```
    prettyPrintMatrix(m); //9.2 appears on top of 2nd column
    m.identity(); //reset matrix
    m.appendTranslation(16, 19, 21);
    prettyPrintMatrix(m); //the translation factors appear in column 4
  }
  protected function prettyPrintMatrix(m:Matrix3D):void {
    var str:String = "";
    for (var col:int = 0; col < 4; col++) {
      var line:String = "|";
      for (var row:int = 0; row < 4; row++) {
        line += "\t" + m.rawData[row*4+col].toPrecision(3) + "\t";
      }
      str += line + "|\n";
    }
    trace(str);
  }
 }
}
```

These methods enable some matrix operations on a `Matrix3D` instance:

- `identity():void` — Replaces the matrix with a 4×4 identity matrix.

- `transpose():void` — Replaces the matrix with its transpose. Transposing a matrix swaps its columns and rows. It's easiest to think of this as flipping a matrix (as you'd write it down on paper) across the top-left-to-bottom-right diagonal. For a matrix **M,** the transpose is written $\mathbf{M}^T$. Also, $(\mathbf{M}^T)^T = \mathbf{M}$.

```
| 1 2 3 |ᵀ   | 1 4 7 |
| 4 5 6 | = | 2 5 8 |
| 7 8 9 |    | 3 6 9 |
```

- `invert():Boolean` — Replaces the matrix with its inverse. Inverting a matrix is discussed earlier in "Matrices." Not all matrices are invertible, but all transformation matrices are. The method aborts and returns `false` if the matrix is not invertible.

- `determinant:Number` — A read-only property that calculates the determinant of the matrix. If this is zero, the matrix is not invertible.

Unlike 2D transformations, in 3D you have the option of applying the transformation to the transformation matrix before all the existing transformations or after all the existing transformations. Recall that the `concat()` method of `Matrix` multiplied its argument to the contents of the matrix, with the argument matrix first and the subject matrix (the object whose `concat()` method you call) second. `Matrix3D`, on the other hand, does not have a `concat()` method but instead has `prepend()` and `append()` methods that perform the multiplication with the argument matrix on either the left or

the right hand of the multiplication. Additionally, each transformation matrix method has two flavors. One immediately prepends the matrix and the other immediately appends it, as follows:

- `prependTranslation(x:Number, y:Number, z:Number):void,` `appendTranslation(x:Number, y:Number, z:Number):void` — Composes a translation with the existing transformation. The arguments determine the offsets in each axis.

- `prependScale(xScale:Number, yScale:Number, zScale:Number):void,` `appendScale(xScale:Number, yScale:Number, zScale:Number):void` — Composes a scale with the existing transformation. The arguments determine the scale factor in each direction (1 being no change).

- `prependRotation(degrees:Number, axis:Vector3D, pivotPoint:Vector3D = null):void, appendRotation(degrees:Number, axis:Vector3D, pivotPoint:Vector3D = null):void` — Composes a rotation with the existing transformation. The arguments describe the rotation as a rotation around a certain vector, which need not be one of the coordinate space's axes. When passed a `pivotPoint` in the object's own coordinate space, this position is used for the center of rotation.

There's another convenient way to set the rotation of an object using its transformation matrix. The `pointAt()` method sets the contents of the `Matrix3D` to a transformation that reorients its subject to point at a certain target. This can shortcut potentially painful calculations, and it's quite convenient for games. Example 34-6 creates a vector pointing in a random direction, uses `Vector3D` methods you just learned to set its magnitude to the radius of a sphere. This algorithm creates a random point on the surface of a sphere, because the surface of a sphere is the union of all points the same distance from its center. Then `pointAt()` is used to reorient an arrow to point at the randomly placed target.

EXAMPLE 34-6 `http://actionscriptbible.com/ch34/ex6`

Matrix3D's pointAt() Method

```
package {
  import flash.display.*;
  import flash.events.MouseEvent;
  import flash.geom.Matrix3D;
  import flash.geom.Vector3D;
  [SWF(width="500",height="500",backgroundColor="#000000")]
  public class ch34ex6 extends Sprite {
    protected const targetSphereRadius:Number = 180;
    protected var holder:Sprite;
    protected var target:Shape;
    protected var arrow:Shape;
    public function ch34ex6() {
      holder = new Sprite();
      holder.x = stage.stageWidth/2; holder.y = stage.stageHeight/2;
      addChild(holder);
      makeArrow();
      makeTarget();
```

EXAMPLE 34-6 *(continued)*

```
      stage.addEventListener(MouseEvent.CLICK, pointToRandom);
      pointToRandom(null);
    }
    protected function makeTarget():void {
      var SIZE:Number = 10;
      target = new Shape();
      target.graphics.beginFill(0xff0000, 0.8);
      target.graphics.drawCircle(0,0, SIZE);
      holder.addChild(target);

      holder.graphics.lineStyle(0, 0xffffff, 0.8);
      holder.graphics.drawCircle(0, 0, targetSphereRadius);
    }
    protected function makeArrow():void {
      var SIZE:Number = 20, ARROW_SIZE:Number = 9, LINE_WIDTH:Number = 4;
      arrow = new Shape();
      arrow.graphics.lineStyle(LINE_WIDTH, 0xffffff, 1, false,
        LineScaleMode.NORMAL, CapsStyle.NONE, JointStyle.MITER, ARROW_SIZE);
      arrow.graphics.moveTo(-SIZE, 0);
      arrow.graphics.lineTo(SIZE-LINE_WIDTH/2, 0);

      arrow.graphics.moveTo(SIZE-ARROW_SIZE, ARROW_SIZE);
      arrow.graphics.lineTo(SIZE, 0);
      arrow.graphics.lineTo(SIZE-ARROW_SIZE, -ARROW_SIZE);
      holder.addChild(arrow);
    }
    protected function pointToRandom(event:MouseEvent):void {
      var r:Function = function():Number {return Math.random() - 0.5;}
      var randomDirection:Vector3D = new Vector3D(r(), r(), r());
      randomDirection.normalize(); //sets the magnitude to 1
      randomDirection.scaleBy(targetSphereRadius); //sets the magnitude to r
      m = new Matrix3D();
      m.position = randomDirection;
      target.transform.matrix3D = m;

      var m:Matrix3D = new Matrix3D();
      m.pointAt(randomDirection, Vector3D.X_AXIS, Vector3D.Z_AXIS);
      arrow.transform.matrix3D = m;
    }
  }
}
```

In Example 34-6, I passed several arguments to `pointAt()`. Now look at the signature to see what they do.

```
    function pointAt(pos:Vector3D, at:Vector3D = null, up:Vector3D = null):void
```

The first argument, `pos`, is the location that you want the display object — the subject of this transformation matrix — to point at. Optionally, `at` defines the orientation of the display object in its own

coordinate space, where it's pointing by default. I drew an arrow facing to the right, so its orientation is along the +x axis. Optionally, `up` defines which direction is "up" to the display object in its own coordinate system. Because I drew the arrow using 2D vector graphics, it's a drawing facing you on the xy plane. "Up" points out from the screen toward you, so "up" in the drawing of the arrow is the −z axis. I've passed the +z axis, but it doesn't make much of a difference because the back of the arrow looks identical to the front of the arrow. (Tricky!)

The example also uses the `position` vector of a `Matrix3D`. This property gives another way to modify the translation part of the matrix. Unlike `prependTranslation()` and `appendTranslation()`, though, modifying this sets the translation anew rather than concatenating the results.

There are some other methods that let you deal with the transformation matrix as a sum of its constituent parts (translation, rotation, scale). Because the scale and rotation matrices use some of the same matrix elements, it's not fun to try to pick apart a matrix that combines all these kinds of transformation. Thankfully, Flash Player does it for you and even lets you put these constituents back together.

- `decompose(orientationStyle:String = "eulerAngles"):` `Vector.<Vector3D>` — Pulls apart a transformation matrix into a `Vector` of three vectors, which represent its translation, rotation, and scale, in that order. As you might guess, the translation is stored in a three-dimensional vector as $<t_x, t_y, t_z>$, and the scale likewise as $<s_x, s_y, s_z>$. The `orientationStyle` you pass in determines how the method returns the rotation part of the transformation. These are described in the earlier "Orientation" section. Possible values include `Orientation3D.AXIS_ANGLE`, in which case a 4D vector of the form $<axis_x, axis_y, axis_z, \theta>$ is returned; `Orientation3D.EULER_ANGLES`, in which case a 3D vector of the form $<\theta_x, \theta_y, \theta_z>$ aka <pitch, yaw, roll> is returned; and `Orientation3D.QUATERNION`, in which case a 4D vector of the form <x, y, z, w> is returned.

- `recompose(components:Vector.<Vector3D>, orientationStyle:String = "eulerAngles"):Boolean` — (Re)constructs a transformation matrix out of its constituent transformations. The `components` argument expects a `Vector` of three vectors, representing the desired translation, rotation, and scale, in that order, and defined in the manner described earlier. The method returns `false` if you fail to pass these three vectors (because the contents of a `Vector` can't be checked at compile time).

Two important side effects of this pair of methods have to do with how they enable rotation. For one, the methods are the only way to access the rotation of a transformation matrix as quaternions. For another, they enable more convenient access to the rotation as Euler angles. Because `appendRotation()` and `prependRotation()` take the orientation in axis-angle form, you can still do a rotation with Euler angles by calling them three times with the axis set to the x, y, and z axes, and the angles as the orientation's Euler angles. Of course, the `rotationX`, `rotationY`, and `rotationZ` properties also enable direct access to the rotation as Euler angles, if you want to break back out of the transformation API.

`Matrix3D`, like `Matrix`, lets you test the effect of transformation matrices. Of course, whereas `Matrix` (2D) transformed `Points` (2D), `Matrix3D` transforms `Vector3D`s.

- `transformVector(v:Vector3D):Vector3D` — Transforms a single vector with the transformation matrix by multiplying the matrix with the vector (as a column matrix).

- `deltaTransformVector(v:Vector3D):Vector3D` — Transforms a single vector, ignoring any translation.

- `transformVectors(vin:Vector.<Number>, vout:Vector.<Number>):void` — Batch-transforms a set of coordinates. Rather than vectors, these are passed as a list of coordinates, assumed to proceed in an x, y, z, x, y, z pattern. This is useful to transform a whole mesh, perhaps for use with `drawTriangles()`. (See Chapter 40.) You must pass a resizable or appropriately sized `Vector` to `vout` to store the results.

As you can see, the facilities provided for matrix math and 3D geometry calculations are quite robust in ActionScript 3.0. Using transformation matrices directly instead of the `DisplayObject` properties described in Chapter 15 opens up a powerful set of tools.

Combined with some methods in the `Graphics` object and a few classes that haven't been touched on in the `flash.geom` package, Flash Player contains a pretty strong foundation for a 3D graphics pipeline. Read on to Chapter 40 to see more of what's possible.

Summary

- Each display object has a `transform` property from which you can access the `matrix`, `colorTransform`, and `matrix3D` properties to apply transforms to the object.
- Using the `matrix` property, you can apply transforms such as rotation, translation, scaling, and shearing.
- Using the `colorTransform` property, you can apply colors and tints to display objects.
- Using 3D matrices and vectors along with the `matrix3D` property, you can have powerful numerical control over 3D transformations.

Programming Vector Graphics

Drawing programmatically allows you much greater freedom in creating and modifying your graphics at runtime, enabling you to create rich and exciting interactivity. It also enables you to make incredibly small files without embedded graphics — all the drawing executed at runtime by ActionScript. Vector graphics are drawn with lines and fills. Like vector graphics imported into your SWF, they look good at any scale. When programming vector graphics, you have many of the same tools in code that you may be familiar with from drawing software packages like Adobe Illustrator or Inkscape, such as curved and straight lines, rectangles, and ellipses.

Flash Player 10 adds new vector drawing methods that let you efficiently batch drawing commands, draw in three dimensions, and use shaders. The new APIs provide a great low-level toolkit for building three-dimensional graphics engines. Look for the Version notes that indicate features only available in Flash Player 10 and later.

Overview

This chapter needs to cover a lot of ground. A high-level view of the drawing API helps you keep oriented as you learn about the details of each part.

The classes involved in the drawing API are part of the `flash.display` package. Primarily, you'll be using the `Graphics` class, which acts as a canvas for drawing vector graphics. The `Graphics` class is an abstract class; you can't create instances of it directly with `new Graphics()`. Furthermore, it's not a `DisplayObject`, so you can't add it to the display list. To get your vector drawings on stage, you use the existing `Graphics` instances that are attached to all your favorite display objects: `Sprite`, `MovieClip`, and `Shape`. Each of these classes has a public property called `graphics` that you can access to draw programmatically into the instance.

You haven't used much of the Shape class up until now, but it's only now that it steps into the spotlight. Shapes are lightweight display objects that make perfect containers for drawing but do little else. They can't contain other display objects; they don't extend DisplayObjectContainer. Furthermore, they can't receive user interaction like clicks; they don't extend InteractiveObject. But they do extend DisplayObject, so they can be added to a display list. Creating a Shape is the next best thing you can do next to creating a Graphics object. Nonetheless, in examples I'll frequently use Sprite instead; it's good to have user interaction, and the extra overhead is not worth worrying about.

Example 35-1 offers a quick preview of using Graphics to get access to the drawing API and getting those graphics on-screen.

EXAMPLE 35-1 http://actionscriptbible.com/ch35/ex1/

Using the Graphics Class

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  public class ch35ex1 extends Sprite {
    public function ch35ex1() {
      var s:Shape = new Shape();

      s.graphics.lineStyle(3, 0xff0000);
      s.graphics.beginFill(0xffff00);
      s.graphics.drawCircle(0, 0, 30);
      s.graphics.endFill();

      s.x = stage.stageWidth / 2;
      s.y = stage.stageHeight / 2;
      addChild(s);
    }
  }
}
```

Notice that all the drawing API methods are called on the Graphics instance that belongs to the Shape. Alternatively, because the example class extends Sprite, you can use the graphics property to draw directly inside the example class ch35ex1, without need for a child Shape.

Note

I contrast vector drawing with bitmap drawing in this part of the book, but sometimes there is overlap. You can use bitmap data to fill in vector drawings, and you can use the drawing API in Flash Player 10 to output texture-mapped triangles, where the texture maps are bitmaps. So to clarify, the drawing API, the topic of this chapter, includes methods that affect a Graphics object. Bitmap drawing includes any methods that affect a BitmapData object. I'll use BitmapData in some parts of this chapter, but I'll save the in-depth coverage for Chapter 36, "Programming Bitmap Graphics." ∎

The drawing API is implemented as a *state machine*. The drawing functions that you call describe a series of steps that the drawing API follows sequentially. Every drawing function you call changes the state of the drawing API. It's a lot like giving orders: the drawing API carries out the commands it's

given in the order you give them. Just try reading Example 35-1 shown earlier, and imagine you're talking to someone who has a pen. First, set the pen color to red. Then start coloring things in yellow. Draw a circle of radius 30, and finish up. As you see more complex examples, you realize that the order of drawing commands is vitally important.

Note
If you're around my age, you no doubt used a simplistic programming language called Logo to move a little turtle around the screen, telling it to `RIGHT 90` **and** `FORWARD 100` **and** `RIGHT 90` **and** `FORWARD 100`**. If you know what I'm talking about, congratulations: you're old. Also, you know what a state machine is.**

If you've ever programmed in OpenGL, that's a state machine, too. Think of the Flash Player drawing API as OpenGL's dull third cousin. ∎

I'll start by covering the most atomic drawing operations: moving the "pen," drawing line segments, drawing curve segments, and filling them in. Then I'll cover styles — how the lines and fills look. After that I'll discuss primitives that you can draw with one call, like Example 35-1.

Flash Player 10 and later let you codify a bunch of drawing commands in objects, throw them into a `Vector`, and pass the whole package to a `Graphics` object, rather than writing each command as ActionScript code. This lets you do some interesting metaprogramming: although you can't have your program change its own code, you can certainly have it change a list of drawing commands. The bulk of this chapter focuses on the procedural style of using the drawing API; once you're well grounded in the API, I'll cover the command equivalents of the functions you learned, and then you'll see how to batch them and execute them. It's quite easy to bridge these two ways of using the API.

Armed with this overview, you can now start drawing vector graphics with the Flash Player drawing API.

Drawing Basics

You can think of the `Graphics` object as a robot holding a pen. The robot is pretty dumb; you have to tell it every little thing it should do. Pretend you're just learning how to use your robot. Test it and see if it works. Maybe you can have it move the pen around.

Moving the Pen

To move the pen, use the `moveTo()` command. Remember that all these methods are defined on the `Graphics` class. To specify where to move the pen, just give the robot x and y coordinates. Keep in mind that these x and y coordinates are in the coordinate space of the `Graphics` object; if you rotate and scale the display object that owns the `Graphics` canvas, you'll see your drawings rotate and scale, but all your drawing commands continue to be relative to the original coordinate space. This, too, applies to the whole drawing API. Got it?

```
function moveTo(x:Number, y:Number):void
```

Now test it.

```
var s:Shape = new Shape();
s.graphics.moveTo(10, 10);
s.graphics.moveTo(20, 20);
s.graphics.moveTo(30, 30);
```

If you try out this code, you'll see ... nothing. You can't really verify that the robot is moving the pen, because, of course, the robot is pure analogy.

The drawing API uses the same coordinate system as the rest of the Flash Player API: the origin is at the top left, x increases to the right, and y increases down.

Straight Line Segments

Want to see if you can get the robot to produce something worth looking at? How about a line segment? Line segments are finite, straight lines that connect two points. Now, because this is a state machine, a simpleton robot, you don't tell it which two points to connect, but rather "put your pen down and move the pen to ...." The robot faithfully draws the line from *where the pen was last* to *where you tell it to go*. That's one reason you need moveTo().

```
function lineTo(x:Number, y:Number):void
```

Why don't you try drawing an X? Planning how you'd tell the robot to move, you could move the pen to the top left, place the pen down and draw a line to the bottom right, pick up the pen and move it to the top right, and then put the pen down again and draw a line to the bottom left. Example 35-2 shows one way you could do it. (Of course, you could do the strokes in any order or direction. This is merely one option that seems natural to me because of the way I was trained to write. The robot could easily draw bottom to top or a combination of directions.)

EXAMPLE 35-2 `http://actionscriptbible.com/ch35/ex2`

Drawing Line Segments

```
package {
  import flash.display.Sprite;
  public class ch35ex2 extends Sprite {
    public function ch35ex2() {
      graphics.lineStyle(1);
      graphics.moveTo(0, 0);
      graphics.lineTo(20, 20);
      graphics.moveTo(20, 0);
      graphics.lineTo(0, 20);
    }
  }
}
```
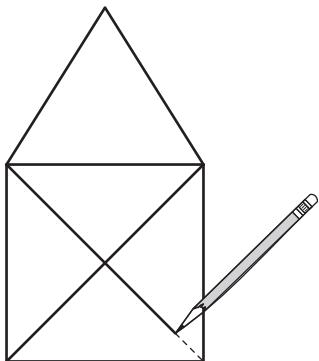
Notice anything tricky about this? I've snuck in two things. First, I'm drawing inside the example class because it's a Sprite and contains a Graphics instance in its graphics property. Second, I've cheated and set a line style. Line and fill styles deserve their own section — it's coming right up. But if you want to see any lines, you have to set a line style. If you comment out the first line in the example, you won't see anything. So I've issued one additional command to the drawing robot: "Use a pen that's one pixel thick." There are a ton more settings for the line, but I've let the rest keep their defaults.

You don't have to pick up the pen between every line segment like this example does. Remember the exercise of seeing if you can draw a little house (see Figure 35-1) without lifting the pencil or crossing lines? Is the draw-bot up to the challenge? Example 35-3 has the answer.

FIGURE 35-1

Drawing lines without lifting the pencil

EXAMPLE 35-3 `http://actionscriptbible.com/ch35/ex3`

Drawing Connected Lines

```
package {
  import flash.display.Sprite;
  public class ch35ex3 extends Sprite {
    public function ch35ex3() {
      graphics.lineStyle(2);
      graphics.moveTo(0, 100);
      graphics.lineTo(0, 50);
      graphics.lineTo(25, 0);
      graphics.lineTo(50, 50);
      graphics.lineTo(0, 50);
      graphics.lineTo(50, 100);
      graphics.lineTo(50, 50);
      graphics.lineTo(0, 100);
      graphics.lineTo(50, 100);
      x = y = 10;
    }
  }
}
```

Before going on, I want to try something fun. Example 35-4 shows drawing a Hilbert curve in two dimensions. A *Hilbert curve* is a fractal curve that follows a simple rule of drawing forward and turning to describe an open cup like a "U" . . . but if you recurse and follow the same rules between the turns,

you'll come up with a winding path that fills the screen. After a few iterations, it's kind of hard to believe that the dense tapestry it creates is the same single line, but nowhere do you "pick up the pen." Figure 35-2 shows the curve after a few iterations.

EXAMPLE 35-4 `http://actionscriptbible.com/ch35/ex4`

Drawing a Continuous Fractal Curve

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  public class ch35ex4 extends Sprite {
    protected var iterations:int = 0;
    protected var hilbertCurve:Hilbert;

    public function ch35ex4() {
      hilbertCurve = new Hilbert(stage.stageWidth, stage.stageHeight);
      addChild(hilbertCurve);
      stage.addEventListener(MouseEvent.CLICK, onClick);
      onClick(null);
    }

    protected function onClick(event:MouseEvent):void {
      if (++iterations > 7) {
        iterations = 1;
        hilbertCurve.scaleX = hilbertCurve.scaleY = 1;
      }
      hilbertCurve.clear();
      hilbertCurve.hilbert(iterations);
      hilbertCurve.scaleY = hilbertCurve.scaleX /= 2;
    }
  }
}
import flash.display.Shape;
class Hilbert extends Shape {
  protected const U:int = 0, R:int = 1, D:int = 2, L:int = 3;
  protected var curx:Number, cury:Number;
  protected var len:Number;

  public function Hilbert(w:Number, h:Number) {
    len = Math.min(h, w);
  }

  public function clear():void {
    graphics.clear();
    graphics.lineStyle(0);
    cury = 0;
    curx = len/2;
    graphics.moveTo(curx, cury);
  }
```

```
    public function hilbert(iter:int, dir:int = U):void {
      if (--iter < 0) return;
      switch(dir) {
        case L: //left-facing cup is drawn right, down, left
          hilbert(iter, U);
          drawSegment(R);
          hilbert(iter, L);
          drawSegment(D);
          hilbert(iter, L);
          drawSegment(L);
          hilbert(iter, D);
          break;
        case R: //right-facing cup is drawn left, up, right
          hilbert(iter, D);
          drawSegment(L);
          hilbert(iter, R);
          drawSegment(U);
          hilbert(iter, R);
          drawSegment(R);
          hilbert(iter, U);
          break;
        case U: //up-facing cup is drawn down, right, up
          hilbert(iter, L);
          drawSegment(D);
          hilbert(iter, U);
          drawSegment(R);
          hilbert(iter, U);
          drawSegment(U);
          hilbert(iter, R);
          break;
        case D: //down-facing cup is drawn up, left, down
          hilbert(iter, R);
          drawSegment(U);
          hilbert(iter, D);
          drawSegment(L);
          hilbert(iter, D);
          drawSegment(D);
          hilbert(iter, L);
          break;
      }
    }

    protected function drawSegment(dir:Number):void {
      switch (dir) {
        case U: graphics.lineTo(curx, cury -= len); break;
        case R: graphics.lineTo(curx += len, cury); break;
        case D: graphics.lineTo(curx, cury += len); break;
        case L: graphics.lineTo(curx -= len, cury); break;
      }
    }
  }
}
```
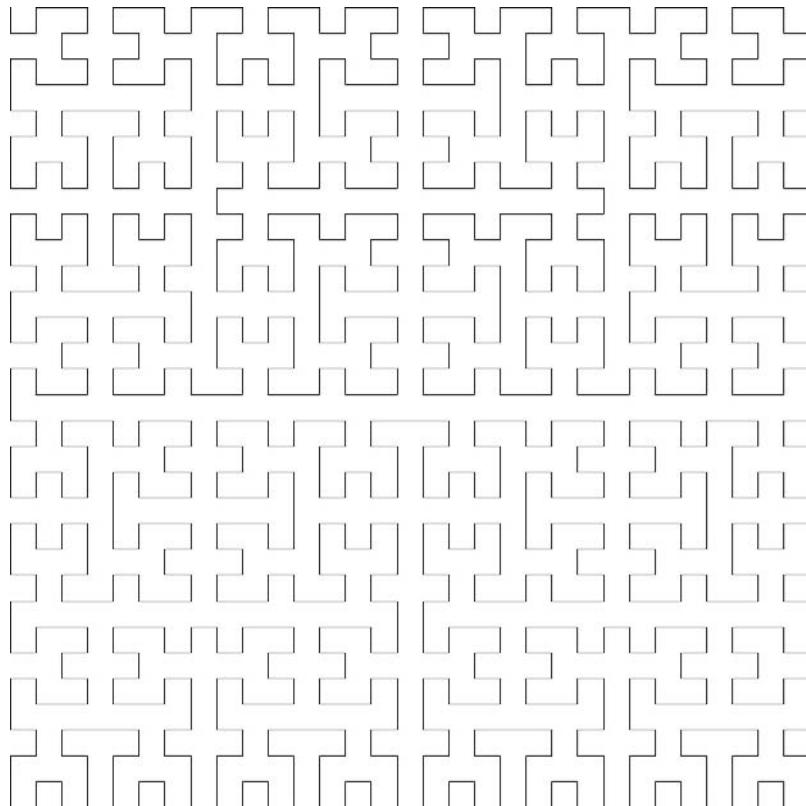
FIGURE 35-2

A Hilbert curve drawn with lineTo()

Curved Line Segments

But enough boring straight lines. How about drawing some pretty curves? The friendly draw-bot draws a curve with the following method:
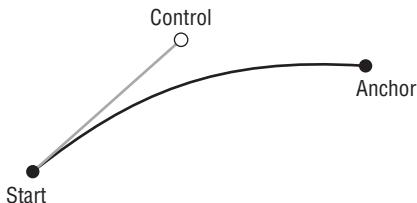
```
function curveTo(controlX:Number,
                 controlY:Number,
                 anchorX:Number,
                 anchorY:Number):void
```

Now, it looks like only two points are encoded in this method call, but remember that commands are relative to the current location of the "pen." So three points are involved in a curve: the initial position, a control point, and an endpoint. The curve draws from the initial position to the endpoint at (`anchorX`, `anchorY`). It does not pass through the control point at (`controlX`, `controlY`) but bends toward it. This is illustrated in Figure 35-3.

This kind of curve is a *quadratic Bézier* curve. The kind of curves you may be used to drawing in illustration software packages are *cubic Bézier* curves. These have two control points, or "handles," per segment. You'll notice that the curveTo() method uses only one control point.

FIGURE 35-3

A curve and its control point

Maybe you want to draw a longer, continuous curve. There are a plethora of techniques for approximating other kinds of curves with quadratic Béziers. However, you can create perfectly natural continuous curves with a simple technique.

Your first thought might be to continue the curve by attaching segments to each other, with the endpoint of one being the starting point of the next. Before you raise your hand to answer what's wrong with this, I'll show you. You can try Example 35-5, clicking to add points to the would-be curve. (Press any key to clear them.)

EXAMPLE 35-5 `http://actionscriptbible.com/ch35/ex5`

Continuous Curves, First Attempt

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.events.KeyboardEvent;
  import flash.events.MouseEvent;
  import flash.geom.Point;
  public class ch35ex5 extends Sprite {
    protected var pts:Vector.<Point>;
    protected var curve:Shape;
    protected var points:Shape;
    public function ch35ex5() {
      points = new Shape();
      addChild(points);
      curve = new Shape();
      addChild(curve);
      //click to add a point
      stage.addEventListener(MouseEvent.CLICK, onClick);
      //press any key to clear the screen
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
      onKeyDown(null);
    }
    protected function onClick(event:MouseEvent):void {
      var p:Point = new Point(stage.mouseX, stage.mouseY);
      pts.push(p);
      drawCurve();
```

continued

695

EXAMPLE 35-5 *(continued)*

```
      //the points you originally clicked will be displayed
      points.graphics.drawCircle(p.x, p.y, 2);
    }
    protected function onKeyDown(event:KeyboardEvent):void {
      pts = new Vector.<Point>();
      curve.graphics.clear();
      points.graphics.clear();
      points.graphics.lineStyle(1, 0, 0.5);
    }
    protected function drawCurve():void {
      if (pts.length < 3) return;
      curve.graphics.clear();
      curve.graphics.lineStyle(1);
      curve.graphics.moveTo(pts[0].x, pts[0].y);
      for (var i:int = 0; i < pts.length - 2; i += 2) {
        curve.graphics.curveTo(pts[i+1].x, pts[i+1].y, pts[i+2].x, pts[i+2].y);
      }
    }
  }
}
```

If you play with this code, you'll quickly see the problem. The good thing is, the resulting curve is continuous: you connected the endpoint of every quadratic curve segment to the control point and endpoint of the next segment. However, its first derivative is not continuous. In other words, the slope changes abruptly between segments. Right now the curve segments are completely independent. You haven't done anything to make sure the curves blend into each other. The solution I'm going to propose is a simple one. It may not be desirable because the curve it produces doesn't actually go through any of the points you give it — but it looks nice and it's trivial.

Perhaps you have a bunch of points to define a curve. First draw midpoints between every set of two points. You've just created a bunch of new points. Now, instead of drawing curves only using the initial set of points, you have interpolated points to draw with, as you can see in Figure 35-4. If you use the interpolated points as endpoints and the original points as control points, the result is a curve that's both continuous and smooth, as Example 35-6 shows.

EXAMPLE 35-6 http://actionscriptbible.com/ch35/ex6

Continuous Curves, Second Attempt

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.events.KeyboardEvent;
  import flash.events.MouseEvent;
  import flash.geom.Point;
```
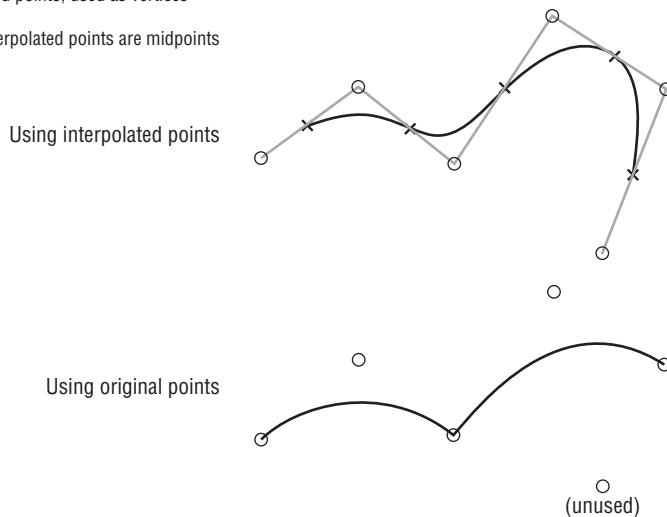
```
public class ch35ex6 extends Sprite {
  protected var pts:Vector.<Point>;
  protected var curve:Shape;
  protected var points:Shape;
  public function ch35ex6() {
    points = new Shape();
    addChild(points);
    curve = new Shape();
    addChild(curve);
    //click to add a point
    stage.addEventListener(MouseEvent.CLICK, onClick);
    //press any key to clear the screen
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    onKeyDown(null);
  }
  protected function onClick(event:MouseEvent):void {
    var p:Point = new Point(stage.mouseX, stage.mouseY);

    //first add in the midpoint between the last point and this point
    if (pts.length > 0)
    {
      var midpoint:Point = Point.interpolate(pts[pts.length-1], p, 0.5);
      pts.push(midpoint);
      points.graphics.lineStyle(1, 0, 0.2);
      points.graphics.lineTo(p.x, p.y);
      points.graphics.drawCircle(midpoint.x, midpoint.y, 5);
    }
    pts.push(p);
    points.graphics.lineStyle(1, 0, 0.5);
    points.graphics.drawCircle(p.x, p.y, 2);

    drawCurve();
  }
  protected function onKeyDown(event:KeyboardEvent):void {
    pts = new Vector.<Point>();
    curve.graphics.clear();
    points.graphics.clear();
  }
  protected function drawCurve():void {
    if (pts.length < 4) return;
    curve.graphics.clear();
    curve.graphics.lineStyle(3);
    curve.graphics.moveTo(pts[1].x, pts[1].y);
    //throw out the first point (i = 1) and the last (i < pts.length - 3)
    //because you're only drawing between midpoints
    for (var i:int = 1; i < pts.length - 3; i += 2) {
      curve.graphics.curveTo(pts[i+1].x, pts[i+1].y, pts[i+2].x, pts[i+2].y);
    }
  }
}
}
```

FIGURE 35-4

Quadratic Bézier curve segments, using unmodified set of points and using interpolated anchor points

○   Points in set

✕   Interpolated points, used as vertices

—   Shows interpolated points are midpoints

Using interpolated points

Using original points

○
(unused)

This technique — midpoint interpolation — is useful for generating smooth curves, but not for approximating higher-degree Bézier curves. The math behind Bézier curves is really cool and surprisingly graspable. It would be easy to keep going, but there's a lot more drawing API to cover! I highly recommend reading the articles and TechNotes on Jim Armstrong's site (`http://algorithmist.wordpress.com/` and `http://algorithmist.net/technotes .html`), which are a treasure trove of curvy goodness. Also, the Wikipedia entry on Béziers is not half bad (`http://en.wikipedia.org/wiki/Bezier_curve`).

Filling a Path

You can instruct the drawing API to fill the path you define. Most likely, you're going to want to ensure you draw closed paths while using fills so that you know where your fill is filling. Because the analogy robot has such a terrible memory, you have to tell it when to start filling the paths it's drawing and when to stop. If you forget to tell the little guy to stop, he goes along his merry way, spilling ink all over as he keeps drawing paths you never intended to be filled. So remember, for every begin fill command, you must issue an end fill command!

Now, these commands, like line styles, have different commands for different kinds of fills. You'll learn about those next, but for now stick with solid-color fills. You can start filling with a solid color using this method:

```
function beginFill(color:uint, alpha:Number = 1.0):void
```

Finish any kind of fill with the simple method:

```
function endFill():void
```

Just because you should close your paths when using fills doesn't mean you have to. Flash Player fills the path you define as if you had connected the first and the last point on it.

Have some fun with another example. Recall that you can define a circle parametrically as

$$x(t) = \cos(t)$$
$$y(t) = \sin(t)$$

This means that you can draw a circle by drawing a point for every value of t in its range of 0 to 2π. If you start a fill, draw points for a bunch of values of t, and end the fill, you should be able to draw a circle.

But circles are boring! Who cares! You'll see how to make a circle in one line later. For now, you'll learn how to tweak things by bubbling the edge of the circle up. You do this by modifying each x and y value by adding in another term that describes a more rapidly rotating circle. Imagine the Earth circling the sun and tracing a circle (fine, ellipse, if you have to be like that). Now imagine the moon circling the Earth more rapidly. Try to trace out in your mind what the path of the moon relative to the sun looks like. That's what you'll do in Example 35-7.

EXAMPLE 35-7 http://actionscriptbible.com/ch35/ex7

Filling a Path

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.utils.getTimer;
  [SWF(backgroundColor="#000000", frameRate="45")]
  public class ch35ex7 extends Sprite {
    protected var r:Number; //radius
    protected var shape:Shape;
    public function ch35ex7() {
      shape = new Shape();
      shape.x = stage.stageWidth / 2;
      shape.y = stage.stageHeight / 2;
      addChild(shape);
      r = Math.min(stage.stageWidth, stage.stageHeight) * 0.3;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      var w:Number = int(30 * ((stage.mouseX / stage.stageWidth) - 0.5));
      var r2:Number = 2 * ((stage.mouseY / stage.stageHeight) - 0.5)
      r2 = r * 3 * Math.pow(r2, 3);
      shape.graphics.clear();
      shape.graphics.beginFill(0xff00cc);
      for (var t:Number = 0; t < Math.PI * 2; t += 0.03) {
        var x:Number = r * Math.cos(t) + r2 * Math.cos(t * w);
        var y:Number = r * Math.sin(t) + r2 * Math.sin(t * w);
        if (t == 0) shape.graphics.moveTo(x, y);
        shape.graphics.lineTo(x, y);
      }
      shape.graphics.endFill();
    }
  }
}
```

Note the `beginFill()` call before the sequence of `lineTo()` calls and the `endFill()` after it. If you run this example, control the amount of pucker or bloat — the radius of the moon's orbit — with the y position of your mouse, and control the number of moon orbits per Earth orbit with the x position of your mouse. You'll notice that for small pucker and bloats, the fill is solid because the deviation from a smooth circle is small. Once the values get out of hand, however, the path you ask the drawing API to fill crosses over itself. When this happens, the outside edge of the path enters the circle itself, so a lacuna is formed. The fill, in addition to closing gaps, doesn't fill where the path twists inside itself.

Clearing Graphics

Well, thanks a lot, Example 35-7 — the cat's out of the bag. There's not much more to clearing the graphics context than calling the method

```
function clear():void
```

This method also resets any line or fill styles.

Setting Drawing Styles

Now, you've drawn some interesting graphics, but you've done so without style. No style, no grace, no panache, no guts! The drawing API allows you to use the following for both strokes and fills:

- Solid colors
- Gradients
- Bitmaps
- Shaders (Pixel Bender content)

Version

Flash Player 10 and later support all listed content for both strokes and fills. Flash Player 9 supports solid-color, gradient, and bitmap fills and solid-color and gradient strokes. ∎

Solid Colors

Solid colors are a simple way to get graphics on-screen quickly and easily. Color fills and strokes may have an alpha value; keep in mind that you can apply any filters, blend modes, and transformations to the display objects that contain your `Graphics` object. So plain colors aren't all that bad.

To start a solid-color fill, use

```
function beginFill(color:uint, alpha:Number = 1.0):void
```

The `color` is a standard RGB unsigned `int` such as `0xff0000` (red). Set a transparency with the optional parameter `alpha`, which should be scaled between 0 and 1.

To set a solid-color stroke style, use this monster method:

```
function lineStyle(thickness:Number = NaN,
                   color:uint = 0,
                   alpha:Number = 1.0,
                   pixelHinting:Boolean = false,
                   scaleMode:String = "normal",
                   caps:String = null,
                   joints:String = null,
                   miterLimit:Number = 3):void
```

As you can see, there are plenty of options, but all of them are optional. Much of the time, the first three parameters will suffice, but it's great to have all the options available. They give you an insane amount of control over how strokes are drawn.

- `thickness` — Thickness of the stroke in pixels. Strokes are drawn centered on the path that defines them. This parameter can range from 0 to 255. If `NaN` (Not a Number, see Chapter 7, "Numbers, Math, and Dates") is passed, no stroke is drawn. You can also turn off strokes by calling `lineStyle()` with no arguments, because the default value for `thickness` is `NaN`. A value of 0 uses *hairline* thickness: no matter how the graphics' display object is scaled, the strokes draw one pixel thick.

- `color` — Color of the stroke.

- `alpha` — Opacity of the line, between 0 and 1.

- `pixelSnapping` — When `true`, instructs the stroke to snap to whole-pixel values. Defaults to `false`. See the following examples.

- `scaleMode` — How the thickness of the line scales. By default, the value is `"normal"` or `LineScaleMode.NORMAL`, which means the line scales as the graphics' display object scales. For example, a 1-pixel line within a `Shape` that is scaled by a factor of 2 appears 2 pixels wide. Using the value `LineScaleMode.NONE` prevents the stroke from scaling under any circumstance. Using `LineScaleMode.VERTICAL` *prevents* the stroke from scaling when the display object is only scaled vertically. If the stroke is scaled horizontally or a combination, the stroke scales despite using `LineScaleMode.VERTICAL`. The same applies for `LineScaleMode.HORIZONTAL` and horizontal-only scaling.

- `caps` — The kind of cap to apply to the stroke's end. Options are `CapsStyle.NONE`, `CapsStyle.ROUND`, and `CapsStyle.SQUARE`. The default is round caps. See the following examples to compare cap styles.

- `joints` — How lines join to one another. If two lines share a common endpoint, Flash Player applies a joint style. Options are `JointStyle.BEVEL`, `JointStyle.MITER`, and `JointStyle.ROUND`. By default, the joint style is round. See the following examples to compare joint styles.

- `miterLimit` — Only used with the miter joint specifies the distance from the vertex at which the joint is mitered or cut off. Expressed as a factor of the thickness, so that a value of 3 indicates that the joint may extend three times the thickness of the stroke past the vertex.

Example 35-8 illustrates pixel snapping. Pixel snapping tells the anti-aliasing code in Flash Player to draw strokes that are close to being on a whole pixel on the pixel boundary rather than blending between the neighboring pixels. The code draws two identical rectangles just slightly off whole pixels. The only difference is that one uses pixel snapping in the line style, and the other does not. When you test the code, you notice that the first rectangle has smoothed-out lines that show the inaccuracy, whereas the second appears sharp and straight. The effect is shown in Figure 35-5.

EXAMPLE 35-8 `http://actionscriptbible.com/ch35/ex8`

Pixel Snapping

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  public class ch35ex8 extends Sprite {
    public function ch35ex8()
    {
      var nosnap:Shape = new Shape();
      addChild(nosnap);
      nosnap.graphics.lineStyle(0, 0, 1, false);
      nosnap.graphics.lineTo(100, 0.4);
      nosnap.graphics.lineTo(100.4, 50);
      nosnap.graphics.lineTo(0, 50.4);
      nosnap.graphics.lineTo(0.4, 0)
      nosnap.x = nosnap.y = 20;

      var snappy:Shape = new Shape();
      addChild(snappy);
      snappy.graphics.lineStyle(0, 0, 1, true);
      snappy.graphics.lineTo(100, 0.4);
      snappy.graphics.lineTo(100.4, 50);
      snappy.graphics.lineTo(0, 50.4);
      snappy.graphics.lineTo(0.4, 0);
      snappy.x = 200; snappy.y = 20;
    }
  }
}
```
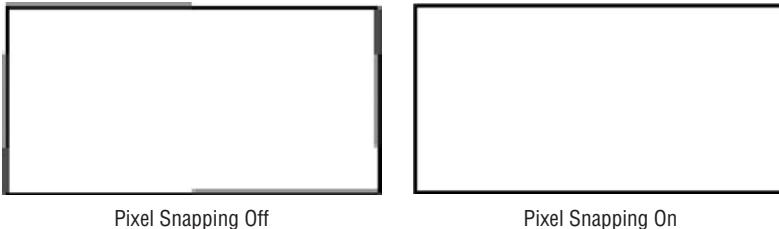
Example 35-9 illustrates the scale type parameter. Both rectangles are scaled to four times their original size in the x direction. However, whereas the line thickness in the first rectangle scales, the line thickness in the second does not. The effect is shown in Figure 35-6.

Notice that for the special case of 1-pixel-wide strokes, setting the stroke to hairline thickness has the same effect as setting the `scaleMode` to `LineScaleMode.NONE`.

The pixel-snapping parameter urges edges close to a pixel boundary to draw on the pixel rather than anti-aliasing.

Pixel Snapping Off　　　　　　　　Pixel Snapping On

EXAMPLE 35-9　http://actionscriptbible.com/ch35/ex9

Scale Types

```
package {
  import flash.display.LineScaleMode;
  import flash.display.Shape;
  import flash.display.Sprite;
  public class ch35ex9 extends Sprite {
    public function ch35ex9() {
      //normal scale mode (by default), strokes scale
      var normalscale:Shape = new Shape();
      addChild(normalscale);
      normalscale.graphics.lineStyle(1);
      normalscale.graphics.drawRect(10, 10, 50, 50);
      normalscale.scaleY = 4;

      //no-scale mode, strokes don't scale
      var noscale:Shape = new Shape();
      addChild(noscale);
      noscale.x = 100;
      noscale.graphics.lineStyle(1, 0, 1, false, LineScaleMode.NONE);
      noscale.graphics.drawRect(10, 10, 50, 50);
      noscale.scaleY = 4;

      //normal scale mode, hairline stroke, strokes stay at 1 px wide
      var hairline:Shape = new Shape();
      addChild(hairline);
      hairline.x = 200;
      hairline.graphics.lineStyle(0);
      hairline.graphics.drawRect(10, 10, 50, 50);
      hairline.scaleY = 4;
    }
  }
}
```

FIGURE 35-6

Setting the scale type parameter determines how strokes scale.

1px stroke
Normal scale mode, scaled up

1px stroke
No scale mode, scaled up

Hairline stroke
Normal scale mode, scaled up

Example 35-10 shows the available types of stroke end caps. The effect is shown in Figure 35-7.

EXAMPLE 35-10 http://actionscriptbible.com/ch35/ex10

Stroke End Caps

```
package {
  import flash.display.*;
  public class ch35ex10 extends Sprite {
    public function ch35ex10() {
      var g:Graphics = graphics;
      g.lineStyle(40, 0, 0.5, false, LineScaleMode.NONE, CapsStyle.ROUND);
      drawStroke(40);
      g.lineStyle(40, 0, 0.5, false, LineScaleMode.NONE, CapsStyle.SQUARE);
      drawStroke(100);
      g.lineStyle(40, 0, 0.5, false, LineScaleMode.NONE, CapsStyle.NONE);
      drawStroke(160);
    }
    protected function drawStroke(y:Number):void {
      graphics.moveTo(0, y);
      graphics.lineTo(150, y);
      graphics.lineStyle(0);
      graphics.moveTo(0, y);
      graphics.lineTo(150, y);
    }
  }
}
```
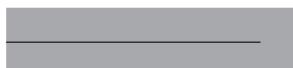
The line cap styles

CapsStyle.ROUND

CapsStyle.SQUARE

CapsStyle.NONE

Example 35-11 illustrates the join styles and miter limits. The effect is shown in Figure 35-8.

`http://actionscriptbible.com/ch35/ex11`

Joins and Miters
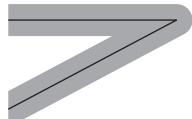
```
package {
  import flash.display.*;
  public class ch35ex11 extends Sprite {
    public function ch35ex11() {
      graphics.lineStyle(20, 0, 0.3, false, null, null, JointStyle.ROUND);
      drawStrokes(40);
      graphics.lineStyle(20, 0, 0.3, false, null, null, JointStyle.BEVEL);
      drawStrokes(120);
      graphics.lineStyle(20, 0, 0.3, false, null, null, JointStyle.MITER);
      drawStrokes(200);
      graphics.lineStyle(20, 0, 0.3, false, null, null, JointStyle.MITER, 20);
      drawStrokes(280);
    }
    protected function drawStrokes(y:Number):void {
      graphics.moveTo(0, 0+y);
      graphics.lineTo(100, 0+y);
      graphics.lineTo(-20, 50+y);
      graphics.lineStyle(0);
      graphics.moveTo(0, 0+y);
      graphics.lineTo(100, 0+y);
      graphics.lineTo(-20, 50+y);
    }
  }
}
```
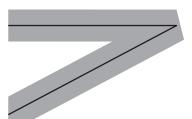
FIGURE 35-8
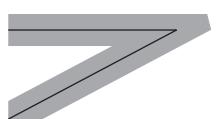
Joint styles
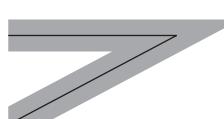
JoinStyle.ROUND

JoinStyle.BEVEL

JoinStyle.MITER, default miter

JoinStyle.MITER, 20px miter

Here I used a tiny shortcut. The string constant parameters to this method may be sent `null`, and they take their default values.

Gradients

The drawing API lets you use gradients for both fills and strokes. The drawing API creates linear and radial gradients for you. When you use a gradient line style, it merely defines how the gradient fills up the stroke. A call to `lineStyle()` is still required to set the basic parameters of the stroke. Once the basic line style is set, the drawing API knows how to draw the shape of the stroke; the gradient, bitmap, and shader line styles merely fill in the shape of the stroke that `lineStyle()` defines.

The methods to apply gradients to strokes and fills are very much the same, so I'll show them as a set.

```
function beginGradientFill(type:String,
                          colors:Array,
                          alphas:Array,
                          ratios:Array,
                          matrix:Matrix = null,
                          spreadMethod:String = "pad",
                          interpolationMethod:String = "rgb",
                          focalPointRatio:Number = 0):void

function lineGradientStyle(type:String,
                          colors:Array,
                          alphas:Array,
                          ratios:Array,
                          matrix:Matrix = null,
```

```
                              spreadMethod:String = "pad",
                              interpolationMethod:String = "rgb",
                              focalPointRatio:Number = 0):void
```

These methods control everything about the gradient, from the colors it's made up of to how they blend into each other. You'll note that at least the first four parameters are required.

- `type` — Whether the gradient is to be *linear* (the color changes gradually along a line) or *radial* (the color changes gradually from a central point and moves outward). Options are `GradientType.LINEAR` and `GradientType.RADIAL`.

- `colors` — An array of color values specified as RGB `uints`. You can specify two or more colors; the gradient blends between these. For linear gradients, the colors gradate from left to right. For radial gradients, the colors gradate from the center out.

- `alphas` — For each color value, you must include a corresponding alpha value. This array parallels the `colors` array, specifying a series of alpha values between 0 and 1.

- `ratios` — The drawing API also needs to know how to weight the colors. Where along the spectrum of the gradient should Flash center each color from the colors array? An array of values from 0 to 255 that correspond to the `colors` and `alphas` arrays specifies this. A value of 0 means that the corresponding color's center should be located at the far left (linear) or center (radial) of the gradient. A value of 255 indicates that the corresponding color's center should be located at the far right (linear) or outside (radial) of the gradient.

- `matrix` — A transformation matrix to be applied to the gradient. This is used to rotate and translate the gradient within the area to be filled or stroked by the gradient. This parameter is optional, but you'll usually need to include it.

- `spreadMethod` — How the gradient should fill up the space provided. Options are `SpreadMethod.PAD` (the default), `SpreadMethod.REFLECT`, and `SpreadMethod.REPEAT`. The `PAD` option fills any leftover space of the line or shape with the color from the edge of the gradient closest to the empty space. `REFLECT` mirrors the gradient repeatedly to fill the space. When the `REPEAT` option is selected, the same gradient simply repeats as necessary. This option results in discontinuities if the colors at the edges of the gradient are not the same.

- `interpolationMethod` — Determines how the intermediate colors in a gradient are calculated. Defaults to `InterpolationMethod.RGB`; this interpolates values in the sRGB color space, which gives a more perceptually smooth gradient. You may also choose `InterpolationMethod.LINEAR`; this interpolates the colors linearly without conversion, which gives a numerically accurate gradient.

- `focalPointRatio` — Used in radial gradients only, this shifts the focus (the center of the gradient) from left (−1) to right (1) within the ellipse.

That's a lot of parameters, so it'll be good to look at a few examples. First, however, you'll learn how to construct a matrix using the `Matrix` class. You can construct a standard `Matrix` object and then use the `createGradientBox()` method. The `createGradientBox()` method accepts up to five parameters — width, height, rotation in radians, amount to translate in the x direction, and amount to translate in the y direction. The `width` and `height` parameters are required. The remaining parameters have default values of 0. The following code creates a `Matrix` object that you can use with the `lineGradientStyle()` method to apply a gradient that has dimensions of 200 by 100 pixels:

```
var mxBox:Matrix = new Matrix();
mxBox.createGradientBox(200, 100);
```

Example 35-12 uses both gradient strokes and gradient fills for a truly gaudy appearance.

707

EXAMPLE 35-12   http://actionscriptbible.com/ch35/ex12

Gradient Strokes and Fills

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  public class ch35ex12 extends Sprite {
    public function ch35ex12() {
      stage.addEventListener(MouseEvent.CLICK, onClick);
    }
    protected function onClick(event:MouseEvent):void {
      var s:Sticker = new Sticker();
      s.x = stage.mouseX;
      s.y = stage.mouseY;
      s.scaleX = s.scaleY = 0.5 + Math.random()*0.5;
      s.rotationZ = (Math.random() - 0.5) * 20;
      addChild(s);
    }
  }
}
import flash.display.*;
import flash.text.*;
import flash.filters.*;
import flash.geom.Matrix;
class Sticker extends Sprite {
  protected static const labels:Array = ["OMG!", "WOW!", "BETA", "MFJ", "ORLY",
    "LULZ", "OMFG", "KITTENS", "ZOMG", "WUT?", "2.0", "SNACKS!", "TACO",
    "ONOES!", "DESU"];
  protected static const SIZE:Number = 100;
  public function Sticker() {
    var gradientMatrix:Matrix = new Matrix();
    gradientMatrix.createGradientBox(SIZE, SIZE, Math.PI/2);

    graphics.lineStyle(5, 0, 1, false, null, null, JointStyle.MITER, 10);
    graphics.lineGradientStyle(GradientType.LINEAR,
      [0xfff600, 0x837749],
      [1, 1],
      [0, 255],
      gradientMatrix
    );

    graphics.beginGradientFill(GradientType.LINEAR,
      [makeColor(), makeColor()],
      [1, 1],
      [0, 255],
      gradientMatrix
    );

    drawPolyStar();
    graphics.endFill();

    var label:TextField = makeLabel();
```

```
      label.x = -0.5 * label.textWidth;
      label.y = -0.5 * label.textHeight;
      addChild(label);
      filters = [new DropShadowFilter(0, 0, 0, 0.6, 64, 64, 1, 2)];
    }
    protected function drawPolyStar():void {
      var rdelta:Number = 10; //how pointy the points are
      var tdelta:Number = Math.PI*2 / 40; //40 points
      for (var t:Number = 0; t < Math.PI*2; t += tdelta) {
        rdelta *= -1;
        var x:Number = (SIZE + rdelta) * Math.cos(t);
        var y:Number = (SIZE + rdelta) * Math.sin(t);
        if (t == 0) {
          graphics.moveTo(x, y);
        } else {
          graphics.lineTo(x, y);
        }
      }
    }
    protected function makeLabel():TextField {
      var tf:TextField = new TextField();
      tf.width = tf.height = 0;
      tf.selectable = false;
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.defaultTextFormat = new TextFormat("_sans", 35, 0xffffff, true, true);
      tf.text = labels[Math.floor(Math.random() * labels.length)];
      return tf;
    }
    protected function makeColor():uint {
      var rnd:Function = function():uint{return uint(Math.random() * 128 + 128)};
      return (rnd() << 16 | rnd() << 8 | rnd());
    }
  }
}
```

First, check out the gradient styles in the `Sticker()` constructor. Notice that I set a stroke style before setting the gradient style. Using a long miter lets the points of the star come to a sharp point. The two gradients are linear gradients, sharing the same matrix transformation that makes them vertical. The stroke is populated with a predetermined gold color, whereas the fill gradient has its colors set at runtime by `makeColor()`. Refer to Chapter 13, "Binary Data and ByteArrays," for more information on the bit math. Notice also that the `rotationZ` property is being used to rotate a device font. This only works in Flash Player 10 and later — but that has nothing to do with the gradient fills and strokes.

You might also notice that the algorithm to draw the star shape is similar to that used in Example 35-7. A starburst shape is basically circular, so I started with the parametric circle equation again. At every point on the starburst, the vertex is either moved in or out from where it should be. In other words, its radius alternates. Every loop, flip the magnitude of a constant added onto the radius of what's otherwise a circle and, voila, you'll get a starburst. Simply make sure that there are an even number of points and that they divide into 2π to ensure that the start and end points of the starburst shape are the same.

I've cheated yet again and snuck in a hint of upcoming chapters. A shadow filter is applied to the stickers to give them depth as they pile up; learn more about filters in Chapter 37.

You can do a lot more with gradients. Try playing with the code to draw other shapes or make more interesting gradients with more colors or radial gradients or with less random color schemes!

Bitmaps

Using the drawing API, you can fill paths with bitmaps. This is particularly cool for tiled graphics. Say you have to fill a huge background area. Instead of loading a 500KB image, you could create a tiling bitmap that's only 10KB and repeat it. With bitmap fills, this task is trivial.

Again, the methods used for bitmap fills and strokes are the same, so look at them as a group.

```
function beginBitmapFill(bitmap:BitmapData,
                         matrix:Matrix = null,
                         repeat:Boolean = true,
                         smooth:Boolean = false):void

function lineBitmapStyle(bitmap:BitmapData,
                         matrix:Matrix = null,
                         repeat:Boolean = true,
                         smooth:Boolean = false):void
```

Version

Flash Player 10 and later support bitmap fills and strokes. Flash Player 9 supports bitmap fills. The `lineBitmapStyle()` method is only available in FP10 and later. ■

The parameters to the bitmap fill and stroke methods are fairly simple. Be sure to read up on bitmap data in Chapter 36.

- `bitmap` — The bitmap data to draw.
- `matrix` — A transformation matrix that allows you to translate and rotate the source bitmap while drawing it into the stroke or fill.
- `repeat` — Set this to `true` to tile the bitmap. Defaults to `true`.
- `smooth` — Set this to `true` to use smoothing when drawing the bitmap. Defaults to `false`. You probably want to keep this off as long as no transformations are applied, especially with pixel tiles.

In Example 35-13, you'll add a bitmap tile that will continually resize itself to match the stage's size. This is perfect for backgrounds in sites. I've used this very class in multiple client projects.

EXAMPLE 35-13 `http://actionscriptbible.com/ch35/ex13`

Tiling a Pattern with Bitmap Fills

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.net.URLRequest;
```

```
    import flash.sampler.NewObjectSample;
    import flash.system.LoaderContext;
    public class ch35ex13 extends Sprite {
      public function ch35ex13() {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        var l:Loader = new Loader();
        l.load(new URLRequest("http://actionscriptbible.com/files/plaid.gif"),
          new LoaderContext(true)); //thanks to squidfingers.com for the tile!
        l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoadComplete);
      }
      protected function onLoadComplete(event:Event):void {
        var b:Bitmap = LoaderInfo(event.target).content as Bitmap;
        if (b) {
          var tile:TileBitmap = new TileBitmap(b.bitmapData);
          addChild(tile);
          tile.followStageSize(stage);
          tile.addEventListener(MouseEvent.CLICK, onClick);
        }
      }
      protected function onClick(event:MouseEvent):void {
        //go full screen on click to see more of this beautiful plaid
        stage.displayState = (stage.displayState == StageDisplayState.NORMAL)?
          StageDisplayState.FULL_SCREEN : StageDisplayState.NORMAL;
      }
    }
}
import flash.display.*;
import flash.events.Event;
class TileBitmap extends Sprite
{
  protected var srcBitmap:BitmapData;
  protected var smoothing:Boolean = false;
  protected var _width:Number = 0;
  protected var _height:Number = 0;
  public function TileBitmap(srcBitmap:BitmapData, smoothing:Boolean = false) {
    super();
    this.smoothing = smoothing
    this.srcBitmap = srcBitmap;
    redraw();
  }
  public function followStageSize(s:Stage):void {
    s.addEventListener(Event.RESIZE,
      function(event:Event):void {matchStageSize(Stage(event.target))});
    matchStageSize(s);
  }
  protected function matchStageSize(s:Stage):void {
    width = s.stageWidth;
    height = s.stageHeight;
  }
```

continued

711

EXAMPLE 35-13   *(continued)*

```
  override public function set width(w:Number):void {
    if (Math.abs(w - width) < 1) return;
    _width = w;
    super.width = _width;
    redraw();
  }
  override public function get width():Number {return _width;}
  override public function set height(h:Number):void {
    if (Math.abs(h - height) < 1) return;
    _height = h;
    super.height = _height;
    redraw();
  }
  override public function get height() : Number {return _height;}
  protected function redraw():void {
    scaleX = scaleY = 1;
    if (srcBitmap && width > 0 && height > 0) {
      graphics.clear();
      graphics.beginBitmapFill(srcBitmap, null, true, smoothing);
      graphics.drawRect(0, 0, Math.ceil(width), Math.ceil(height));
      graphics.endFill();
    }
  }
}
```

After the tile loads (a whopping 1KB!), you can click the background to see it full screen. It should tile magnificently to cover any area.

Game developers can also make good use of `beginBitmapFill()`. It's a fast and simple method for *blitting*, which is just a fancy word for dumping bitmap data onto the screen as fast as possible. Blitting to a graphics surface using the drawing API has far less overhead than creating `Bitmap` instances for each game sprite, though there are definitely other approaches. This is one of those hybrid vector-bitmap programming techniques I mentioned in the overview.

Shaders

Finally, the drawing API lets you fill and stroke paths with output from a shader. You'll learn more about shaders in Chapter 38, "Writing Shaders with Pixel Bender." The methods for using a shader on a stroke and a fill are practically identical:

```
function beginShaderFill(shader:Shader, matrix:Matrix = null):void
```

```
function lineShaderStyle(shader:Shader, matrix:Matrix = null):void
```

Version

FP10. Shaders are only available in Flash Player 10 and up, and so too these methods. ∎

When using these methods, all inputs to the shader must already be satisfied. The transformation matrix, as usual, allows a final linear transformation to be applied before drawing the fill or stroke. Just load in a precompiled Pixel Bender kernel for now, and save the shader code for Chapter 38. In Example 35-14, I've used Ricardo Cabello's plasma shader to stroke a Lissajous curve. *Lissajous curves* are a fun family of parametric equations that let you see harmonic oscillations; circles and ellipses are some of the degenerate cases of a Lissajous curve. The parametric equations are

$$x(t) = \sin(at + d)$$
$$y(t) = \sin(bt)$$

You can see how they are similar to circles. The equations become especially familiar when the d term is $\pi/2$, because $\sin(t + \pi/2) = \cos(t)$.

EXAMPLE 35-14 `http://actionscriptbible.com/ch35/ex14`

Shaders and Lissajous Curves

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.net.*;
  import flash.utils.ByteArray;
  import flash.utils.getTimer;
  [SWF(backgroundColor="#000000")]
  public class ch35ex14 extends Sprite {
    protected var plasma:Shader;
    protected var shape:Shape;
    public function ch35ex14() {
      plasma = new Shader();
      shape = new Shape();
      shape.x = stage.stageWidth/2; shape.y = stage.stageHeight/2;
      addChild(shape);
      var loader:URLLoader = new URLLoader();
      loader.dataFormat = URLLoaderDataFormat.BINARY;
      //plasma shader courtesy the inimitable mrdoob (Ricardo Cabello)
      loader.load(new URLRequest(
        "http://actionscriptbible.com/files/mrdoob-plasma.pbj"));
      loader.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    protected function onLoadComplete(event:Event):void {
      var loader:URLLoader = URLLoader(event.target);
      plasma.byteCode = ByteArray(loader.data);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      var t:int = getTimer();
      var w:Number = stage.stageWidth/2; var h:Number = stage.stageHeight/2;
      var d:ShaderData = plasma.data;
      d.center.value = [Math.sin(t*.002)*300+w, Math.cos(t*.0001)*300+h];
      d.wave.value = [Math.sin(t*.001)*.06+0.01, Math.cos(t*.00005)*.05];
      d.offset.value = [Math.sin(t*.004)*200, Math.cos(t*.0003)*200];
      d.color_offset.value = [Math.sin(t*.002)*2, Math.sin(t*.00009)*2,
        Math.cos(t*.00005)*2];
```

continued

713

EXAMPLE 35-14 *(continued)*

```
    d.distort.value = [Math.sin(t*.0008)*.05];

    shape.graphics.clear();
    shape.graphics.lineStyle(6, 0, 1, false, null, CapsStyle.NONE);
    shape.graphics.lineShaderStyle(plasma);
    drawLissajous();
  }
  protected function drawLissajous():void {
    var H:Number = stage.stageHeight, W:Number = stage.stageWidth;
    var a:Number = ((stage.mouseX / W) - 0.5) * 16;
    var b:Number = ((stage.mouseY / H) - 0.5) * 16;
    for (var t:Number = 0; t < Math.PI * 2; t += 0.02) {
      var x:Number = W/2 * Math.sin(a * t + Math.PI/2);
      var y:Number = H/2 * Math.sin(b * t);
      if (t == 0) {
        shape.graphics.moveTo(x, y);
      } else {
        shape.graphics.lineTo(x, y);
      }
    }
  }
}
}
```

Filling a rectangular area with the result of a shader is one of the quicker ways to get shader output onto the screen.

Drawing Primitives

The drawing API lets you draw any number of simple shapes to the `Graphics` object. Using these built-in primitive drawing methods is much less painful than describing the primitives in terms of their constituent lines and curves. To use all the primitive drawing commands, set your line and fill styles, call the primitive drawing function, and then end any fills you started. In the examples thus far, I've already shown some of the primitive drawing methods in use here and there without comment. The next sections explore these methods in detail.

Rectangles and Squares

Use the `drawRect()` function to draw rectangles and squares.

```
function drawRect(x:Number, y:Number, width:Number, height:Number):void
```

The x and y parameters define the top-left point of the rectangle; the width and height parameters remain a mystery to this day. (Kidding.) Rectangles have a ridiculous number of uses. Just to name one, combine a filled rectangle with a TextField and a bevel filter for a sweet-looking button.

Circles and Ellipses

You can draw a circle with the — surprise! — drawCircle() method.

```
function drawCircle(x:Number, y:Number, radius:Number):void
```

The x and y parameters define the *center* of the circle, and radius defines its radius. Remember that circles draw from the center out, unlike rectangles. To position a circle via the top-left corner of its bounding box, just add its radius to its center x and y positions.

You can draw ellipses with the drawEllipse() method. Do you sense a pattern here?

```
function drawEllipse(x:Number, y:Number, width:Number, height:Number):void
```

Unlike circles, and like rectangles, ellipses are drawn from the top-left corner of their bounding box, a point which the parameters x and y define. Also note that you set an ellipse's width and height rather than its eccentricity or focal distances.

Rounded Rectangles

The Graphics object has two methods to draw rounded rectangles. The first is the drawRoundRect() method, which creates a rectangle masking an inner ellipse. What you end up seeing is the intersection of the two shapes. You get the straight edges of the rectangle, with the roundness of an ellipse at the corners. See Figure 35-9 for an illustration. The method is as follows:

```
function drawRoundRect(x:Number,
                       y:Number,
                       width:Number,
                       height:Number,
                       ellipseWidth:Number,
                       ellipseHeight:Number = NaN):void
```
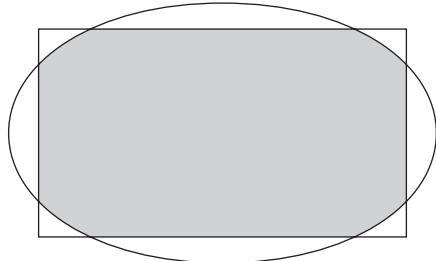
The x and y position define the top-left corner of the rectangle. The width and height parameters determine the size of the rectangle. The size of the ellipse is actually defined relative to the smallest ellipse that fully contains the rectangle, such that the intersection of the two shapes is equal to the rectangle. When you set ellipseWidth and ellipseHeight, you're actually specifying the number of pixels the ellipse should be shrunk in each dimension. The higher these numbers, the smaller the ellipse, and the more the ellipse defines the corner. Or you can simply think of these values as the radius of the corners. You may omit the last parameter, and the ellipse will use the same dimension for ellipseWidth and ellipseHeight.

FIGURE 35-9

The intersection of an ellipse and a rectangle yields a rounded rectangle.

The second way of drawing a rounded rectangle is to use the `drawRoundRectComplex()` method, which allows you to independently define the radius of each of the four corners of your rectangle. This method is way cooler, but it's undocumented, so if you want to be conservative, you might steer clear of it. But at this point the rectangle will boldly go where no rounded rectangle has gone before:

```
function drawRoundRectComplex(x:Number,
                              y:Number,
                              width:Number,
                              height:Number,
                              topLeftRadius:Number,
                              topRightRadius:Number,
                              bottomLeftRadius:Number,
                              bottomRightRadius:Number):void
```

The parameters are pretty self-explanatory. Besides letting you create rectangles with different amounts of rounding on each corner, `drawRoundRectComplex()` is just a lot simpler to think about. On the other hand, it only lets you use circular corners — but I think the elliptical corners tend to look horrendous.

Now you'll put all these primitives to use in a bigger example.

Example: A Drawing Application

Using what you've learned about the drawing API so far, you can create a simple drawing application. This simple application consists of three classes: the `ColorPicker`, the `ToolSet`, and the `DrawingCanvas`. The `DrawingCanvas` contains both the tool set and the color picker, and it uses events to set the color and tool when these are clicked. It also handles mouse interaction, creating a new `curShape` when you press the mouse, and updating it as you move the mouse, depending on the tool, color, and other parameters.

Example 35-15 shows the application, all at once.

EXAMPLE 35-15 http://actionscriptbible.com/ch35/ex15

A Drawing Application

```
package {
  import flash.display.Sprite;
  public class ch35ex15 extends Sprite {
    public function ch35ex15() {
      addChild(new DrawingCanvas(stage));
    }
  }
}
import flash.display.*;
import flash.events.*;
import flash.geom.Point;
import flash.ui.Keyboard;
class DrawingCanvas extends Sprite
{
  private var canvas:Sprite;
  private var toolState:String = "pen";
  private var curShape:Shape;
  private var curColor:uint;
  private var lineThickness:Number = 4;
  private var colorPicker:ColorPicker;
  private var toolSet:ToolSet;
  private var origin:Point = new Point();
  public function DrawingCanvas(_stage:Stage) {
    canvas = new Sprite();
    canvas.graphics.beginFill(0xFFFFFF, 1);
    canvas.graphics.drawRect(0, 0, _stage.stageWidth, _stage.stageHeight);
    canvas.graphics.endFill();
    addChild(canvas);
    _stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    canvas.addEventListener(MouseEvent.MOUSE_DOWN, startDraw);
    canvas.addEventListener(MouseEvent.MOUSE_UP, stopDraw);
    colorPicker = new ColorPicker();
    addChild(colorPicker);
    colorPicker.x = 0;
    colorPicker.addEventListener("colorSelected", changeColor);
    toolSet = new ToolSet();
    addChild(toolSet);
    toolSet.y = 100;
    toolSet.x = 5;
    toolSet.addEventListener("toolSelected", changeTool);
    curShape = new Shape();
  }
private function onKeyDown(event:KeyboardEvent):void {
    switch (event.keyCode) {
    case Keyboard.UP: lineThickness++; break;
```

continued

717

EXAMPLE 35-15   *(continued)*

```
      case Keyboard.DOWN: lineThickness = Math.max(0, lineThickness-1); break;
    }
  }
  private function stopDraw(mouseEvent:MouseEvent):void {
    canvas.removeEventListener(MouseEvent.MOUSE_MOVE, draw);
  }
  private function startDraw(mouseEvent:MouseEvent):void {
    curShape = new Shape();
    origin.x = canvas.mouseX;
    origin.y = canvas.mouseY;
    addChild(curShape);
    canvas.addEventListener(MouseEvent.MOUSE_MOVE, draw);
    if(toolState == "line" || toolState == "pen" || toolState == "eraser") {
      curShape.graphics.moveTo(mouseEvent.stageX, mouseEvent.stageY);
    }
  }
  private function draw(mouseEvent:MouseEvent):void {
    var cursor:Point = new Point(canvas.mouseX, canvas.mouseY);
    var dist:Point = cursor.subtract(origin);
    switch(toolState) {
      case "pen":
        curShape.graphics.lineStyle(lineThickness, curColor);
        curShape.graphics.lineTo(mouseEvent.stageX, mouseEvent.stageY);
        break;
      case "line":
        curShape.graphics.clear();
        curShape.graphics.lineStyle(lineThickness, curColor);
        curShape.graphics.moveTo(origin.x, origin.y);
        curShape.graphics.lineTo(mouseEvent.stageX, mouseEvent.stageY);
        break;
      case "ellipse":
        curShape.graphics.clear();
        curShape.graphics.beginFill(curColor, 1);
        curShape.graphics.drawEllipse(origin.x, origin.y, dist.x, dist.y);
        break;
      case "rect":
        curShape.graphics.clear();
        curShape.graphics.beginFill(curColor, 1);
        curShape.graphics.drawRect(origin.x, origin.y, dist.x, dist.y);
        break;
      case "roundrect":
        curShape.graphics.clear();
        if (dist.x < 0 || dist.y < 0) break;
        curShape.graphics.beginFill(curColor, 1);
        curShape.graphics.drawRoundRect(origin.x, origin.y, dist.x, dist.y,
          lineThickness, lineThickness);
        break;
      case "eraser":
        curShape.graphics.lineStyle(lineThickness, curColor);
```

```
        curShape.graphics.lineTo(mouseEvent.stageX, mouseEvent.stageY);
        break;
    }
  }
  private function changeTool(evt:DataEvent):void {
    toolState = evt.data;
    if (toolState == "eraser") {
      curColor = 0xffffff;
      toolState = "pen";
      lineThickness = 30;
    } else {
      lineThickness = 4;
    }
  }
  private function changeColor(dataEvent:DataEvent):void {
    curColor = parseInt(dataEvent.data, 16);
    curShape.graphics.lineStyle(lineThickness, curColor);
    if (toolState == "eraser") {
      toolState = "line";
    }
  }
}
import flash.display.*;
import flash.geom.ColorTransform;
import flash.events.MouseEvent;
import flash.events.DataEvent;
class ColorPicker extends Sprite {
  private var k:int = 255;
  public function ColorPicker() {
    for(var i:int = 0; i < 256; i += 48) {
      for(var j:int = 0; j < 256; j += 48) {
        var spr:Sprite = new Sprite();
        spr.graphics.beginFill(0xFFFFFF, 1);
        spr.graphics.drawRect(0, 0, 10, 10);
        spr.graphics.endFill();
        spr.buttonMode = true;
        //apply color transofrm
        var trans:ColorTransform = new ColorTransform();
        var red:uint = i << 16;
        trans.color = red + green + blue;
        var green:uint = j << 8;
        var blue:uint = k;
        spr.transform.colorTransform = trans;
        addChild(spr);
        spr.addEventListener(MouseEvent.CLICK, onColorPicked);
        spr.x = i / 4;
        spr.y = j / 4;
        k -= 48;
      }
    }
  }
```

continued

EXAMPLE 35-15 *(continued)*

```
  private function onColorPicked(mouseEvent:MouseEvent):void {
    var target:Sprite = mouseEvent.target as Sprite;
    var trans:ColorTransform = target.transform.colorTransform;
    var color:uint = trans.color;
    var dataEvent:DataEvent = new DataEvent("colorSelected", false, false,
      trans.color.toString(16));
    dispatchEvent(dataEvent);
  }
}
import flash.display.Sprite;
import flash.events.MouseEvent;
import flash.events.DataEvent;
import flash.text.TextField;
import flash.text.TextFieldAutoSize;
class ToolSet extends Sprite {
  public function ToolSet() {
    var pen:Sprite = makeButton("pen");
    pen.graphics.beginFill(0);
    pen.graphics.drawCircle(10, 10, 2);
    pen.graphics.drawCircle(15, 15, 2);
    pen.graphics.drawCircle(20, 20, 2);
    var ellipse:Sprite = makeButton("ellipse");
    ellipse.graphics.beginFill(0)
    ellipse.graphics.drawCircle(15, 15, 12);
    ellipse.graphics.endFill();
    var square:Sprite = makeButton("rect");
    square.graphics.beginFill(0);
    square.graphics.drawRect(5, 5, 20, 20);
    var roundrect:Sprite = makeButton("roundrect");
    roundrect.graphics.beginFill(0);
    roundrect.graphics.drawRoundRect(5, 5, 20, 20, 8);
    var line:Sprite = makeButton("line");
    line.graphics.lineStyle(4);
    line.graphics.moveTo(10, 10);
    line.graphics.lineTo(20, 20);
    var eraser:Sprite = makeButton("eraser");
    eraser.graphics.lineStyle(4);
    eraser.graphics.moveTo(10, 10);
    eraser.graphics.lineTo(20, 20);
    eraser.graphics.moveTo(20, 10);
    eraser.graphics.lineTo(10, 20);
  }
  private function makeButton(id:String):Sprite {
    var s:Sprite = new Sprite();
    s.name = id;
    s.graphics.beginFill(0, 0);
    s.graphics.lineStyle(0, 0, 0.3);
    s.graphics.drawRect(0, 0, 30, 30);
    s.graphics.endFill();
    s.graphics.lineStyle();
```

```
      s.buttonMode = true;
      s.addEventListener(MouseEvent.MOUSE_DOWN, toolClicked);
      var label:TextField = new TextField();
      label.autoSize = TextFieldAutoSize.LEFT;
      label.text = id;
      label.y = -15;
      s.addChild(label);
      s.y = numChildren * 45;
      addChild(s);
      return s;
    }
    private function toolClicked(event:MouseEvent):void {
      var tool:DisplayObject = event.target as DisplayObject;
      dispatchEvent(new DataEvent("toolSelected", false, false, tool.name));
    }
  }
}
```

Phew! All the exciting drawing code is contained in `startDraw()` and `draw()` of the `DrawingCanvas`. Note that you can use the keyboard to change the thickness of the pen and line tools, and the roundness of the corners of the round rect. This example brings together primitives and basic drawing commands.

Batched Drawing

Beginning in Flash Player 10, the drawing API has been augmented with the ability to draw whole chunks of a drawing at once, rather than calling a long string of functions. Command objects encapsulate drawing commands in a compact form and can be used to pass around whole drawings or complex paths. You'll even notice a speed boost when using these batched drawing methods instead of subsequent calls.

Version

FP10. All methods and classes covered in this section are only available in Flash Player 10 and later. ∎

The drawing API lets you batch generic drawings, complex paths, and strips of triangles. All these drawing methods have their own uses and their own slightly different approaches.

Drawing a Path

If you have a complicated path with dozens of points or curves, instead of calling `moveTo()`, `lineTo()`, and `curveTo()` repeatedly, you can store all the essential data about these paths in a dense format and use the `Graphics` object's `drawPath()` method to execute the path. If you'd like to fill the path, you can surround the `drawPath()` call with `beginFill()` and `endFill()` as you normally would; likewise, you can set stroke styles just as you normally would. The drawing API still acts statefully as always. By drawing a path with `drawPath()`, you merely collapse all the `moveTo()`, `lineTo()`, and `curveTo()` calls into one function call.

721

So what is the "essential data" about paths? Most importantly, you need to know the points on the "canvas" that are involved in the path. You can store a list of *x* and *y* positions used to draw the path. By using a `Vector` of `Numbers` (`Vector.<Number>`), you can store a collection of points by simply alternating *x*, *y*, *x*, *y*, *x*, *y* ... in the `Vector` by convention. This is an outrageously compact way to store a sequence of locations! This can be referred to as the "coordinates vector."

But paths can involve moving the pen, drawing a straight line segment, and drawing a quadratic Bézier segment. Just storing the points is not enough: you have to know what to do with them. You've heard the possible options several times. If you simply assign a unique numeric value to each option and store them in a `Vector` of ints (`Vector.<int>`), you'll have a compact way to know what to do with the locations. This can be called the "commands vector." You'll encode the possible path operations in the manner described in Table 35-1.

TABLE 35-1

Path Commands Enumerated

Int	Const	Command Type
0	`GraphicsPathCommand.NO_OP`	Do nothing. Might be useful if you want to pad your commands vector.
1	`GraphicsPathCommand.MOVE_TO`	A `moveTo()` command. Expects one point (two `Numbers`) in the coordinates vector.
2	`GraphicsPathCommand.LINE_TO`	A `lineTo()` command. Expects one point (two `Numbers`) in the coordinates vector.
3	`GraphicsPathCommand.CURVE_TO`	A `curveTo()` command. Expects an anchor point and an endpoint (four `Numbers`) in the coordinates vector.
4	`GraphicsPathCommand.WIDE_LINE_TO`	A `lineTo()` command, but allows you to use a dummy point that will be ignored, expecting four `Numbers` in the coordinate vector rather than two. Use this and the following command if you want to keep your coordinates vector aligned regularly.
5	`GraphicsPathCommand.WIDE_MOVE_TO`	A `moveTo()` command, but allows you to use a dummy point that will be ignored, expecting four `Numbers` in the coordinate vector rather than two. Use this and the preceding command if you want to keep your coordinates vector aligned regularly.

When drawing a path using `drawPath()`, you use the commands vector and the coordinates vector to encode all the information about a path. There's one more argument that determines the order in which points are drawn, but you'll learn about that later.

These two `Vectors` are interdependent; the commands vector acts like an index for the coordinates vector. As the long-forgotten drawing robot processes a path using `drawPath()`, it has to look at the commands vector to see not only what to do with the next numbers in the coordinates vector, but how many of them to read off of the coordinates vector. You'll notice some interesting operations,

like NO_OP, WIDE_LINE_TO, and WIDE_MOVE_TO. You, the programmer, can use these operations to make some guarantees about the pattern of these lists; this can make it easier to loop through the vectors. For instance, if you only use WIDE_LINE_TO and WIDE_MOVE_TO instead of their "narrow" equivalents, you know that every four Numbers in the Vector defines a new command. You could loop through the vector without paying attention to the commands vector. Or, if you wanted to loop through both vectors with the same index variable, you could pad the commands vector with NO_OPs, perhaps three NO_OPs following every other command along with the "wide" commands, allowing you to iterate through both vectors like so:

```
for (var i:int = 0; i < commandsVector.length; i += 4) {
  var command:int = commandsVector[i]; //the following 3 are padded with NO_OP
  var point0:Point = new Point(coordinatesVector[i], coordinatesVector[i+1]);
  var point1:Point = new Point(coordinatesVector[i+2], coordinatesVector[i+3]);
  //do something with it...
}
```

In most cases, you may find this overkill. These examples use the normal move, line, and curve commands.

Finally, the Method

Look at the drawPath() method. It will be used as an example.

```
drawPath(commands:Vector.<int>,
         data:Vector.<Number>,
         winding:String = "evenOdd"):void
```

The first two arguments are the command vector and the coordinates vector just discussed.

Winding

The winding argument tells the happy draw-bot what to do when paths overlap. In Example 35-7, you saw that when a filled path crosses itself, the fill alternates between present and vacant. This is *even-odd* winding.

If you create overlapping paths that are drawn in a given order, you can determine whether a given intersection is filled by summing up the directions of the paths that intersect: clockwise paths counting as +1 and counterclockwise paths counting as −1. Even-odd winding colors only areas that sum up to odd numbers, leaving even-numbered intersections void. That's why you see an alternating pattern in Example 35-7; the clockwise path crosses itself time and time again, incrementing the winding order by 1 each time. This is both the default for drawPath() and the only way that the nonbatched drawing API commands fill paths. To specify even-odd winding, use the constant GraphicsPathWinding.EVEN_ODD.

The other option to determine how intersecting areas fill is *nonzero* winding. This method, as you might guess, fills all intersecting areas that have a nonzero winding count. In Example 35-7, this fills all areas, because the purely clockwise path crosses itself, and all intersecting areas have a positive winding order of various magnitude. All intersecting areas with a zero winding order are left blank. For example, if you intersect a clockwise path with a counterclockwise one, the intersection is not filled because the winding order is 0. To specify nonzero winding, use the constant GraphicsPathWinding.NON_ZERO.

This chapter doesn't bother with the winding argument in its examples; it's optional.

Example: Drawing a Glyph

A pervasive example of complex paths in use is fonts. Every glyph in a font is defined by a path. When rendering text, your computer follows these paths and fills them in. You can get really creative by moving the rendering of text away from `TextFields` and into the drawing API. For instance, you could bulge the letters, smooth them out, vary them up, or transform them so that they appear to be in three dimensions.

A vector-based 3D library, Five3D by Mathieu Badimon (`http://five3d.mathieu-badimon .com/`), does just this. By storing the outlines of fonts and transforming them in perspective as they are drawn, Five3D allows for perspective vector drawing. Although you can transform objects in 3D in Flash Player 10 and later, Five3D has the advantage of still using the drawing API for these transformed vectors, rather than rasterizing the vectors before applying the perspective transformations, as Flash Player does internally (see Figure 35-10). In any case, one great thing about Five3D is that it comes with a drop-in to Flash that can generate a "typography file" from a font on your computer. The output of this file is just what you need to draw a path, in a startlingly similar format.

Five3D in action

In Example 35-16, you leverage Five3D's font conversion to draw a glyph's path in one triumphant line. To make it a little interesting, you'll use a gradient fill, which you'd have to resort to some trickery to implement with `TextFields`. Of course, there's the chance that by the time you read this, Mathieu will have upgraded Five3D or the font file generator to take full advantage of `drawPath()`.

EXAMPLE 35-16   `http://actionscriptbible.com/ch35/ex16`

Using drawPath()

```
package {
  import flash.display.*;
  import flash.events.KeyboardEvent;
  import flash.geom.Matrix;
  public class ch35ex16 extends Sprite {
    public function ch35ex16() {
      MuseumFoundry.initialize();
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKey);
    }
    protected function onKey(event:KeyboardEvent):void {
      var key:String = (String.fromCharCode(event.charCode).toLowerCase());
      if (!(key in MuseumFoundry.__motifs)) return;

      var pathData:Object = Five3DGlyphUtils.convertGlyph(MuseumFoundry, key);
```

```
        var s:Shape = new Shape();
        var m:Matrix = new Matrix();
        m.createGradientBox(100, 100, Math.PI/2);

        s.graphics.beginGradientFill(GradientType.LINEAR, [0xff0000, 0x0000ff],
          [1, 0.7], [0, 255], m);
        s.graphics.drawPath(pathData.commands, pathData.coordinates);
        s.graphics.endFill();

        s.blendMode = BlendMode.MULTIPLY;
        s.x = stage.mouseX;
        s.y = stage.mouseY;
        addChild(s);
      }
    }
  }
import flash.display.GraphicsPathCommand;
class Five3DGlyphUtils {
  public static function convertGlyph(typo:Object, glyph:String):Object {
    var fiveCommands:Array = typo.__motifs[glyph];
    var commands:Vector.<int> = new Vector.<int>(fiveCommands.length);
    var coordinates:Vector.<Number> = new Vector.<Number>();

    for each (var fiveCmd:Object in fiveCommands) {
      var fiveCmdType:String = fiveCmd[0];
      var fiveCmdCoords:Array = fiveCmd[1];
      switch (fiveCmdType) {
        case "M":
          commands.push(GraphicsPathCommand.MOVE_TO);
          coordinates.push(fiveCmdCoords[0], fiveCmdCoords[1]);
          break;
        case "L":
          commands.push(GraphicsPathCommand.LINE_TO);
          coordinates.push(fiveCmdCoords[0], fiveCmdCoords[1]);
          break;
        case "C":
          commands.push(GraphicsPathCommand.CURVE_TO);
          coordinates.push(fiveCmdCoords[0], fiveCmdCoords[1],
                           fiveCmdCoords[2], fiveCmdCoords[3]);
          break;
      }
    }

    return {commands: commands, coordinates: coordinates};
  }
}

//Museum Foundry font by Raph Levien (http://levien.com)
//licensed under Open Font License
//converted to Five3D Typography class
//Five3D by Mathieu Badimon (http://five3d.mathieu-badimon.com/)
```

EXAMPLE 35-16   *(continued)*

```
class MuseumFoundry {
  static public var __motifs:Object = {};
  static public var __widths:Object = {};
  static public var __heights:Number = 100;
  static public var __initialized:Boolean = false;
  //NOTE: the outlines of the font are embedded in this class file.
  //They are extremely large. Only part of one glyph will be shown
  //to conserve space.
  //Please download or run the example online to see its effect.
  static private function initializeMotifs():void {
    __motifs["a"] = [['M',[22.9,60.2]],['L',[22.7,67.5]],['C',[22.7,67.55...
  }
}
```

Click inside the stage and type letters to draw those letters in Museum, an open font by Raph Levien. The Five3DGlyphUtils class converts paths from the Five3D typography format — which you will notice is the same information you need, in a slightly modified form — into the command vector and coordinate vector objects necessary to draw the entire glyph at once using drawPath().

Because paths are represented as two Vectors, they are extremely compact and easy to serialize and deserialize (convert into a form that can be written to disk or sent over the network). This would be an ideal way to implement a multiuser drawing app or save and repeat user-generated drawings.

Storing a Path in a Command

One part of Example 35-16 that bugs me is that you have to pass the two Vectors in an anonymous Object to get them out of the function. That's not very object oriented! You should be able to encapsulate *all* knowledge about a path into one path object. Enter GraphicsPath.

On one level, GraphicsPath is just a wrapper class for a path defined as you have seen before, with the properties commands, data, and winding defining the properties drawPath() expects. You can set these properties or assign some or all of them at construct time:

```
var p1:GraphicsPath = new GraphicsPath();
p1.commands = commandsVector;
p1.data = coordinatesVector;
var p2:GraphicsPath = new GraphicsPath(commandsVector, coordinatesVector);
```

On another level, GraphicsPath provides an easier way to construct the data that makes up a path. It contains methods to push each kind of path command onto the Vectors: moveTo(), lineTo(), curveTo(), and the "wide" methods wideLineTo() and wideMoveTo(). You call these just like you would call the corresponding methods on a Graphics object, but remember that they're used to modify the path stored in the GraphicsPath, not the "canvas" on-screen. Because these methods write to the vectors internally, you don't have to worry about modifying them manually. An error in writing to these vectors that puts the two out of sync — by adding the wrong number of numbers to the coordinates vector, for instance — ruins the rest of the path. Using the convenience functions prevents this.

Finally, `GraphicsPath` is interesting because it implements `IGraphicsData`, meaning that it's one of the drawing commands that can be drawn in a batch. "But wait," you say, "isn't a path already a batch of moves, lines, and curves?" Indeed, you are right. But because you almost always draw paths with more than one segment, paths are considered one atomic drawing command in a batch. The next section looks at how all the `Graphics` methods translate to `IGraphicsData` batch commands.

To draw a batch of commands, use the `Graphics` object's `drawGraphicsData()` method. This method takes a `Vector` of `IGraphicsData` objects (`Vector.<IGraphicsData>`). Using this method, you can include fill and stroke options in a list of commands. But I'm getting ahead of myself. In Example 35-17, you'll try upgrading the `Five3DGlyphUtils` class to output a `GraphicsPath` object instead of the anonymous object you'd used before. Then you'll use `drawGraphicsData()` rather than `drawPath()` to get the path on-screen.

EXAMPLE 35-17 http://actionscriptbible.com/ch35/ex17

Using GraphicsPath

```
package {
  import flash.display.*;
  import flash.events.KeyboardEvent;
  import flash.geom.Matrix;  public class ch35ex17 extends Sprite {
    public function ch35ex17() {
      MuseumFoundry.initialize();
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKey);
    }
    protected function onKey(event:KeyboardEvent):void {
      var key:String = (String.fromCharCode(event.charCode).toLowerCase());
      if (!(key in MuseumFoundry.__motifs)) return;

      var s:Shape = new Shape();
      var m:Matrix = new Matrix();
      m.createGradientBox(100, 100, Math.PI/2);

      var batch:Vector.<IGraphicsData> = new Vector.<IGraphicsData>();
      batch.push(new GraphicsGradientFill(GradientType.LINEAR,
        [0xff0000, 0x0000ff], [1, 0.7], [0, 255], m));
      batch.push(Five3DGlyphUtils.convertGlyph(MuseumFoundry, key));
      batch.push(new GraphicsEndFill());
      s.graphics.drawGraphicsData(batch);

      s.blendMode = BlendMode.MULTIPLY;
      s.x = stage.mouseX;
      s.y = stage.mouseY;
      addChild(s);
    }
  }
}
import flash.display.GraphicsPathCommand;
import flash.display.GraphicsPath;
class Five3DGlyphUtils {
```

continued

EXAMPLE 35-17   *(continued)*

```
  public static function convertGlyph(typo:Object, glyph:String):GraphicsPath {
    var fiveCommands:Array = typo.__motifs[glyph];
    var path:GraphicsPath = new GraphicsPath();

    for each (var fiveCmd:Object in fiveCommands) {
      var fiveCmdType:String = fiveCmd[0];
      var fiveCmdCoords:Array = fiveCmd[1];
      switch (fiveCmdType) {
        case "M":
          path.moveTo(fiveCmdCoords[0], fiveCmdCoords[1]);
          break;
        case "L":
          path.lineTo(fiveCmdCoords[0], fiveCmdCoords[1]);
          break;
        case "C":
          path.curveTo(fiveCmdCoords[0], fiveCmdCoords[1],
                       fiveCmdCoords[2], fiveCmdCoords[3]);
          break;
      }
    }
    return path;
  }
}

//NOTE: MuseumFoundry font class omitted for space.
```

Note that in addition to the path object, the begin fill and end fill methods have been converted into commands. All these instructions are batched together into one Vector of `IGraphicsData` and submitted in one line using `drawGraphicsData()`.

Batching Generic Drawing Commands

Not just paths, but all the basic drawing API methods can be represented in `IGraphicsData` command objects. A `Vector` of these can be drawn in a batch with the `Graphics` object's `drawGraphicsData()` method, as you just saw. This provides a powerful way to store drawings, modify them with code, serialize and deserialize them and, of course, draw them. In Table 35-2, you'll see how different drawing API calls translate to `IGraphicsData` objects. To make sense of how these objects relate, you can cross-reference the informal class diagram in Figure 35-11.

There are no drawing command objects for `clear()` or any of the primitive-drawing commands such as `drawRect()`.

By storing all the drawing commands in a `Vector` rather than immediately applying them to a `Graphics` object, you gain the ability to modify them with code during runtime. Here's an interesting exercise for you: modify Example 35-15, appending drawing commands to a `Vector`,

and implement undo functionality by popping off the most recent command and redrawing. You may want to disable the primitive drawing tools for simplicity, because they have no corresponding `IGraphicsData` objects.

TABLE 35-2
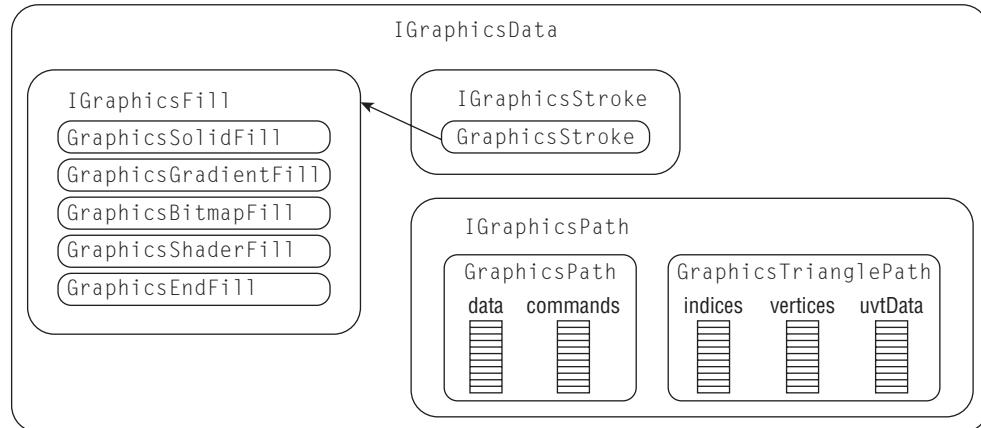
Converting Drawing API Calls to Drawing Command Objects

Drawing API Call	IGraphicsData Command Object	Notes
`moveTo()`	`GraphicsPath`	Add onto the `data` and `commands` vectors, or use `moveTo()` or `wideMoveTo()`.
`lineTo()`	`GraphicsPath`	Add onto the `data` and `commands` vectors, or use `lineTo()` or `wideLineTo()`.
`curveTo()`	`GraphicsPath`	Add onto the `data` and `commands` vectors, or use `curveTo()`.
`drawPath()`	`GraphicsPath`	Encapsulate `data` and `command` vectors, `winding` into a GraphicsPath instance.
`beginFill()`	`GraphicsSolidFill`	Use the same parameters in `GraphicsSolidFill`'s constructor.
`beginGradientFill()`	`GraphicsGradientFill`	Use the same parameters in `GraphicsGradientFill`'s constructor.
`beginBitmapFill()`	`GraphicsBitmapFill`	Use the same parameters in `GraphicsBitmapFill`'s constructor.
`beginShaderFill()`	`GraphicsShaderFill`	Use the same parameters in `GraphicsShaderFill`'s constructor.
`endFill()`	`GraphicsEndFill`	
`lineStyle()`	`GraphicsStroke`	Use the same parameters in `GraphicsStroke`'s constructor, with the addition of an optional `fill` parameter instead of subsequent `line*Style()` call.
`lineGradientStyle()`	`GraphicsGradientFill`	Assign the `GraphicsGradientFill` object to the `GraphicsStroke`'s `fill` property.
`lineBitmapStyle()`	`GraphicsBitmapFill`	Assign the `GraphicsBitmapFill` object to the `GraphicsStroke`'s `fill` property.
`lineShaderStyle()`	`GraphicsShaderFill`	Assign the `GraphicsShaderFill` object to the `GraphicsStroke`'s `fill` property.
`drawTriangles()`	`GraphicsTrianglePath`	Use the same parameters in `GraphicsTrianglePath`'s constructor.

Drawing command objects and their relationships

Copying and Pasting Drawings

In Flash Player 10 and later, the drawing API allows you to easily clone a drawing performed in one `Graphics` object to another. To do so, simply use the `copyFrom()` method, passing in the `Graphics` object to clone. For example:

```
var destination:Graphics, source:Graphics;
destination.copyFrom(source);
```

This method clones the drawing commands performed in one context to the other. When you use the drawing command objects that implement `IGraphicsData`, replicating drawings is painless: just apply the batch of operations using `drawGraphicsData()` as before. But in cases where you haven't kept a record of the commands — where you have a `Graphics` object in an arbitrary, complex state — copying the drawing is the perfect way to duplicate the state.

Note

In Flash Player 9, you can replicate a `Graphics` object by drawing it to a bitmap with `BitmapData.draw()`. However, this doesn't actually clone the drawing; it just saves it as a bitmap. You can learn more about `draw()` in Chapter 36. ∎

Example 35-18 creates a kind of interactive kaleidoscope by copying a drawing into many mirrored shapes.

Cloning Graphics

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

```
public class ch35ex18 extends Sprite {
  protected const NUM_COPIES:int = 6;
  public function ch35ex18() {
    for (var i:int = 0; i < NUM_COPIES; i++) {
      var copy:Shape = new Shape();
      copy.graphics.lineStyle(8, 0xE91C7A, 0.3, false);
      copy.rotationY = 70;
      copy.rotationZ = i * (360 / NUM_COPIES);
      copy.x = stage.stageWidth / 2;
      copy.y = stage.stageHeight / 2;
      addChild(copy);
    }
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
  }
  protected function onMouseMove(event:MouseEvent):void {
    var source:Shape = Shape(getChildAt(0));
    source.graphics.lineTo(stage.mouseX - stage.stageWidth/2,
                           stage.mouseY - stage.stageHeight/2);
    for (var i:int = 1; i < NUM_COPIES; i++) {
      var copy:Shape = getChildAt(i) as Shape;
      copy.graphics.copyFrom(source.graphics);
    }
  }
}
}
```

3D Drawing

The drawing API in Flash Player 10 and later also hides one of the most powerful low-level tools for drawing 3D graphics. Alongside methods that innocuously draw lines, curves, and shapes is a powerful method that can draw texture-mapped triangles to the screen. You can use this as the basis of an engine that draws polygonal, skinned models, or as a one-off effect to distort an image. This method is `drawTriangles()`, with a corresponding graphics command object, `GraphicsTrianglePath`.

Version
FP10. `drawTriangles()` **is only available in Flash Player 10 and later.** ∎

This advanced 3D technique is covered in more depth in Chapter 40, "Advanced 3D."

Summary

- Drawing vector graphics programmatically in Flash Player is performed through the `Graphics` object.
- You can't instantiate a `Graphics` object or put it in the display list. Use the `graphics` property of a `Shape`, `Sprite`, or `MovieClip` instead.

731

- Graphics objects are state machines.
- Basic path operations include moving the "pen," drawing a straight-line segment, and drawing a curve.
- To stroke a path, you must first set a line style with `lineStyle()`.
- Line styles can be followed by stroke fill styles for non-solid-color fills.
- To fill a path, use `begin[`*`kindof`*`]Fill()` methods and `endFill()`.
- Clear the canvas by calling `clear()`.
- Strokes and fills may be styled with solid colors, gradients, bitmap fills, and shader fills (varies by Flash Player version).
- Draw primitives with methods like `drawRect()` and `drawCircle()`.
- Paths can be stored as lists of command types and coordinates and drawn with `drawPath()` (varies by Flash Player version).
- Paths, fills, and strokes can be encapsulated in drawing command objects that implement `IGraphicsData` and drawn with `drawGraphicsData()` (varies by Flash Player version).
- Mapping textures to polygons is possible with `drawTriangles()` (varies by Flash Player version).

Programming Bitmap Graphics

Complementing the vector graphics API described in Chapter 35, "Programming Vector Graphics," Flash Player has an easy-to-use bitmap graphics API. Bitmap graphics are composed of pixels instead of strokes and fills. They are a direct way to work with on-screen graphics. Just as having direct access to binary data opens a whole set of possibilities, direct access to the pixels on-screen means you can display anything you can imagine, or at least anything you can figure out how to code.

Draw bitmap graphics in Flash Player with two classes: `BitmapData` and `Bitmap`. The `BitmapData` class stores image data, but by itself, `BitmapData` doesn't display anything. The `Bitmap` class is a `DisplayObject` that displays the contents of the `BitmapData` object associated with it. In other words, `BitmapData` is the model and `Bitmap` is the view; you need a `Bitmap` to display a `BitmapData`.

In this chapter you'll see many applications of the bitmap drawing API from the mundane to the magical as you learn how to use the `Bitmap` and `BitmapData` classes.

Bitmaps and Their Applications

A bitmap is the simplest and most literal representation of an image. Bitmaps store the color of every pixel in the image. They usually do this in some sort of two-dimensional array, so you can access the color of a given pixel by its location in the matrix. Bitmaps in Flash Player store one 32-bit integer value for each pixel, which stores the color and transparency of the pixel. This color value is the combination of four 8-bit integers, which represent the alpha, red, green, and blue (ARGB) values of the pixel, respectively.

Note

If you extract just one of these color values from every pixel, you end up with a single *channel* of the image. For example, look at the alpha value of every pixel, and you have the alpha channel of that image. A channel is a single coherent data stream within a larger set of data. Bitmaps in Flash Player have alpha, red, green, and blue channels. The same concept applies to video streams that multiplex video and (potentially multiple) audio channels, or stereo audio streams that interleave left and right channels. ∎

In Flash Player, bitmaps are stored in `BitmapData` objects, rather than two-dimensional arrays of integers. This way, `BitmapData` can provide useful methods on top of, and its own interface to, the underlying data.

The bitmap graphics API in Flash Player — which is to say, the collection of classes and methods in the Flash Player API that deal with bitmaps — has one striking property. It's completely integrated into the display list. Why is that so impressive? Video, text, 3D graphics, masked display objects, huge nested compositions, and single objects can all be easily accessed as bitmaps in the same way. The display list makes no distinctions, which means there are no barriers to your creativity. Like-wise, `Bitmap` is a `DisplayObject` like any other and can be placed in the display list without limit. Because the bitmap API is full circle, you can do things like replacing a video with a bitmap that color-corrects the video frames in real time.

Bitmaps are important as an alternate to the display list. If you have a visual idea that's difficult or impossible to accomplish using sets of display objects, you can always fall back on a bitmap. Bitmaps are the ultimate blank canvas — you have total control over every pixel. This no-rules openness is comparable to the advantage you get with `ByteArray`, covered in Chapter 13, "Binary Data and ByteArrays." For example, recall that `Loader` can load, decode, and display image files in JPEG, PNG, or GIF formats. What if you need to display a RAW file from a digital camera? Without `ByteArray` and `Bitmap`, you'd be stuck with what Flash Player provides; with them, limitations are eradicated. You can load the file into a `ByteArray`, decode it, write the bitmap to a `BitmapData`, and display it in a `Bitmap`. Do that, and you've just eclipsed the capabilities of Flash Player!

This example addresses another important thing about bitmaps. They are always stored uncompressed. It's essential to quickly have access to any pixel value, a property that storing the data compressed wouldn't allow for. This means that the data stored in image files like PNGs and JPGs are not bitmaps — but they can be decompressed and the bitmap data can be extracted.

As an alternative to the display list, bitmaps are great for games! Tile engines use small square tex-tures tiled across the screen to easily generate huge terrains, usually for top-down adventure games or strategy games. Texture atlases use one large bitmap to store different frames of animation; moving the area that's drawn from the atlas to the screen provides a computationally cheap way to animate prerendered game sprites. A frame buffer stores a bitmap as it's built, eventually drawing to the screen at once. You can use frame buffers to incorporate elements of the last frame into the current frame, for example, to draw trails, smoke, fire, halos, and so on. All of these old-school game tricks are easily implemented with bitmaps.

One application of bitmaps is important for more than just games. You can use bitmaps to cheaply scale up content to great sizes. For example, drawing a complex scene using the display list at 1600×1200 can really hurt your frame rate, especially if you're using filters (covered in Chapter 37, "Applying Filters"). Instead, you can draw the scene at 640×480, render it to a bitmap, and scale the bitmap up 2.5×. The end result will lose some clarity by being scaled up, but the frame rate is sure to improve drastically. You don't need to use this trick for video or if you've already chosen to use full-screen with a `fullScreenSourceRect`; both use hardware acceleration in the latest versions of Flash Player.

You've just seen some definite advantages of using bitmaps to display graphics. But the fact that you can read from bitmaps is huge, too. Access to the pixels of an image opens the door to image analysis. You can use computer vision algorithms to extract meaningful information out of a webcam feed, for example, to track a user's face, eyes, or tracking markers. You can track the location of a fiducial marker and attach a 3D scene to the marker, overlaid on the video feed, a gimmick called *augmented reality*. Image analysis isn't limited to webcam data, either. An automated testing system can simulate clicks on your program and compare the screen's appearance to what it expects to see.

Creating and Displaying Bitmaps

In this section, you'll get familiar with the `BitmapData` and `Bitmap` classes, with the goal of displaying a bitmap on stage. First you'll create a `BitmapData` instance. You can create `BitmapData` instances that are blank, or you can initialize them from embedded bitmaps.

Creating Empty BitmapData Instances

You can construct a new `BitmapData` object using its constructor, which takes the following parameters, the first two of which are required.

- `width` — Width of the bitmap in pixels.
- `height` — Height of the bitmap in pixels.
- `transparent` — Whether the bitmap background is transparent. Note that bitmaps have an alpha channel regardless of this setting, but when accessing and modifying a bitmap, alpha channels are discarded if this is not set. Optional; defaults to `true`.
- `fillColor` — Initial color of the bitmap, if it's not transparent. Defaults to 0xFFFFFFFF (opaque white).

Note

Color values, in this chapter and elsewhere, are 32-bit `uint`s written in hex notation, in ARGB order. For instance, 0xFF00CED1 is an opaque turquoise color. Its alpha value is 255 (0xFF), red value is 0 (0x0), green value is 206 (0xCE), and blue value is 209 (0xD1).

Occasionally, the alpha value is left out, as in 0x00CED1. In this case, you can assume the alpha value is fully opaque, or 0xFF.

To review hex notation and bit packing, refer to Chapter 7, "Numbers, Math, and Dates," and Chapter 13, "Binary Data and ByteArrays." ■

The size of a bitmap is fixed at construction time. You have to pick a size at the outset, and you can't change the size later on: you'll have to copy the bitmap data into a new, larger bitmap if you need to grow. These examples construct a variety of bitmaps:

```
//create a 500x200 pixel empty, transparent bitmap
var bmpRectangle:BitmapData = new BitmapData(500, 200);
//create a 500x200 pixel opaque, red bitmap
var bmpRectangle:BitmapData = new BitmapData(500, 200, false, 0xFFFF0000);
```

Use the `BitmapData` constructor when you're creating bitmaps from scratch or existing display objects. Most of the time, you'll be filling in the bitmaps with your own contents before displaying them, so you can usually omit the fill color.

735

Creating Instances of Embedded Assets

In Chapter 16, "Working with DisplayObjects in Flash Professional," you saw how to embed bitmap symbols in a SWF. When linked with a class, these produce subclasses of `BitmapData`. When you want to use an embedded bitmap symbol as `BitmapData`, just instantiate the class. Because the asset class extends `BitmapData`, you have to use the `BitmapData` constructor, at minimum including the `width` and `height` parameters. However, you don't actually have to know the width and height of the embedded bitmap. You can pass 0, or any value, to the constructor, and the resulting `BitmapData` subclass will be the correct size, ignoring your arguments.

```
//assuming your asset is linked to assets.FishingCatBitmap
import assets.*;
var bmp:BitmapData = new FishingCatBitmap(0, 0);
trace(bmp.width, bmp.height); //400 400
```

When using assets embedded using the Flex SDK or the `[Embed]` metadata tag, you can skip this step outright. Images embedded with `[Embed]` are turned into subclasses of `BitmapAsset`, a core Flex class that extends `Bitmap`. This means that it can be added directly to the display list. You explored the differing base classes for imported assets in Example 27-4. You can always get the `BitmapData` out of the asset if you need; you'll see how to access a `Bitmap`'s `BitmapData` shortly.

Displaying Bitmaps

Regardless of how you create a `BitmapData` object, it exists only to store the bitmap's data. To render it, you have to attach it to a `Bitmap` and add it to the display list.

First create the `Bitmap`. Its constructor has the following optional arguments:

- `bitmapData` — The bitmap to draw.
- `pixelSnapping` — Whether the bitmap should be drawn at the nearest pixel when placed between pixels. Options include `PixelSnapping.NEVER`, `PixelSnapping.ALWAYS`, and `PixelSnapping.AUTO` (the default, which uses pixel snapping when the image isn't rotated or scaled). Drawing the bitmap at even pixel values results in the fastest rendering and the least fuzziness.
- `smoothing` — Whether to interpolate the bitmap when it's scaled. Defaults to `true`.

You can read or set any of these parameters at any time, so it's not necessary to declare them in the constructor.

A `Bitmap` is meant to display a single bitmap; it can be repurposed to display a different one simply by setting the `bitmapData` parameter. This would also be the way to retrieve the bitmap data from an embedded `BitmapAsset`.

As a `DisplayObject`, a `Bitmap` instance can be rotated, scaled, skewed, and transformed in 3D. Its width and height reflect its current transformation and may be quite different from the original bitmap's dimensions. As a `DisplayObject`, you can also add it to the stage like so:

```
var bmp:BitmapData = new BitmapData(200, 200, false, 0xFFCC00CC);
var bitmap:Bitmap = new Bitmap(bmp);
addChild(bitmap);
```

Bitmap Quality

Bitmap quality is affected most by each `Bitmap`'s `smoothing` parameter, but there are other factors at play. If the stage quality mode (covered in Chapter 14, "Visual Programming with the Display List") is `StageQuality.LOW`, all bitmap anti-aliasing is disabled regardless of the `smoothing` setting. Mipmapping automatically takes place in Flash Player 9.0.60 and later, which increases the quality of a bitmap when it's scaled down.

Disposing Bitmap Data

Releasing the memory held by a `BitmapData` object is just as important as creating it in the first case. Bitmaps, because they're uncompressed, can take up a huge amount of memory (up to 64MB each). The more bitmaps you use, the more trouble you can get into when not properly managing your `BitmapData` instances.

If your application is designed correctly and you remove all references to a `BitmapData` when you're done with it, the AVM2 garbage collector should free its memory automatically, and in a short amount of time.

As a precaution, however, you can `dispose()` bitmap data when you're done with it. This instructs Flash Player to free the memory used to store that data immediately. It does so without removing the `BitmapData` object. The `BitmapData` object becomes invalid and will hopefully be garbage collected soon. Even if it's not, it's no longer the end of the world, because a `BitmapData` without bitmap data is small.

Only call `dispose()` on a `BitmapData` instance when you're good and done with it — ideally at the same time you remove all references to it. After disposal, if you call any substantive methods on the "empty shell" `BitmapData`, you get a runtime error: removing the bitmap data from a `BitmapData` doesn't leave it in good shape to do much of anything.

```
//create enormous bitmap
var hugebmp:BitmapData = new BitmapData(2880, 2880);
//clear out memory. hugebmp is now dead to me
hugebmp.dispose();
```

Capturing and Copying Bitmaps

As much fun as creating cerulean rectangles is, bitmaps are put to better use by modifying existing imagery. You can capture bitmap data from any display object or by copying another bitmap in whole or part.

Copying from Display Objects

The `draw()` method is hands-down the most important method of `BitmapData`. It captures any `DisplayObject` (or portion thereof) as a bitmap. Keeping in mind that `DisplayObjects` can have any number of children and that the stage is also a `DisplayObject`, the `draw()` method can capture individual display objects, groups or layers, or portions of the whole screen.

The `draw()` method accepts any object that implements the `IBitmapDrawable` interface, which includes not only `DisplayObject` but also `BitmapData`, so you could `draw()` from one bitmap into another. Here I'll focus on drawing display objects.

When you capture the bitmap data of a display object using `draw()`, keep in mind that you're creating a static copy of the pixels the display object renders to. The bitmap remains the same as the source changes. Scale the bitmap up enough, and you'll see the individual pixels of the bitmap, even if the original display object can be scaled up without losing fidelity. And, of course, the bitmap is not interactive; clicking on an image of a button doesn't trigger that button. Hopefully, none of this comes as a shock. Just think of `draw()` as taking a snapshot.

The `draw()` method accepts the following parameters. Only the first parameter is required.

- `source` — The `IBitmapDrawable` object to draw.

- `matrix` — A `Matrix` object used to apply a transformation to the display object as it's drawn. This can scale, rotate, move, or skew the image into a desired orientation without changing the `DisplayObject`. See more on transformations in Chapter 34, "Geometric and Color Transformations." By default, no transformation occurs.

- `colorTransform` — A `ColorTransform` object that adjusts the colors as the pixels are copied. See Chapter 34 for more on color transformations. By default, no changes in color are made.

- `blendMode` — A blend mode to apply if drawing the pixels on top of existing bitmap data. Blend modes are specified using values of `BlendMode`, which you can review in Chapter 14.

- `clipRect` — Defines a region of interest in `source` to capture, as a `Rectangle`. Anything falling outside this area is not drawn. By default, as much of `source` is drawn that fits inside the `BitmapData`, starting from the origin of the display object.

- `smoothing` — Whether to interpolate the content when it is drawn. The default is `false`.

You may want to set some of the later arguments without setting other arguments before it. In this case, you can simply pass `null` to the arguments you want to keep default values for. For example, to set only the `blendMode`, you might write this:

```
bmp.draw(source, null, null, BlendMode.ADD);
```

In this snippet, you pass `null` to some arguments and leave others (`clipRect`, `smoothing`) off entirely.

In many cases, only the first argument is necessary. For example, you can take a snapshot of the stage with the following code:

```
var bmp:BitmapData = new BitmapData(stage.stageWidth, stage.stageHeight);
bmp.draw(stage);
```

Better yet, Example 36-1 shows how to use a *sprite sheet* — a single bitmap that contains many individual pictures — to animate a bitmap character. It loads a flat image with a series of frames laid out in a strip and draws from it to display one frame at a time. In the real application, of course, you hide the strip of images and only show the animating bitmap.

`http://actionscriptbible.com/ch36/ex1`

Capturing a DisplayObject with draw()

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  [SWF(frameRate="10")]
  public class ch36ex1 extends Sprite {
    protected const SIZE:Rectangle = new Rectangle(0, 0, 48, 48);
    protected var TOTALFRAMES:int;
    protected var bmp:BitmapData;
    protected var filmstrip:Loader;
    protected var frame:int = 0;
    public function ch36ex1() {
      filmstrip = new Loader();
      //Animation by Derek Yu – www.derekyu.com – used with permission
      filmstrip.load(
        new URLRequest("http://actionscriptbible.com/files/monkey.png"),
        new LoaderContext(true));
      filmstrip.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
      addChild(filmstrip);
      bmp = new BitmapData(SIZE.width, SIZE.height);
      var bitmap:Bitmap = new Bitmap(bmp);
      addChild(bitmap);
      bitmap.y = SIZE.height + 10;
    }
    protected function onLoad(event:Event):void {
      TOTALFRAMES = filmstrip.width / SIZE.width;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      frame = ++frame % TOTALFRAMES;
      //highlight the area you're drawing
      graphics.clear();
      graphics.lineStyle(0, 0xff0000);
      graphics.drawRect(frame*SIZE.width, 0, SIZE.width, SIZE.height);
      //clear the bitmap
      bmp.fillRect(bmp.rect, 0);
      //shift the filmstrip to the left to put the right frame in place
      var shiftTransform:Matrix = new Matrix();
      shiftTransform.translate(-frame * SIZE.width, 0);
      //use SIZE to clip the shifted graphic to the correct position and size
      bmp.draw(filmstrip, shiftTransform, null, null, SIZE);
    }
  }
}
```

This example delves into transformations just a tiny bit, for a good reason. If you try using no transformation matrix and instead move the `sourceRect`, you'll see that the correct frame of animation is displayed, but in its original location. That is to say, it looks like you're moving a mask along a long strip. In fact, you'll only notice this if you enlarge the bitmap `bmp` wide enough to see those frames. So, you use the `matrix` parameter to move the correct frame into place, and you use the `sourceRect` parameter to clip it. Because the bitmap itself is already the correct size, using a `sourceRect` is optional.

Copying from BitmapData Objects

You can copy pixels not only from display objects but also from `BitmapData` objects. You can use the following methods to copy from a `BitmapData` object:

- `draw()` — Draw pixels from a `BitmapData` just as you'd draw them from a `DisplayObject`.
- `clone()` — Return a new `BitmapData` object with an exact copy of the bitmap.
- `copyPixels()` — Copy a region of a bitmap.
- `copyChannel()` — Copy a single channel of another bitmap, or copy one channel into another.
- `merge()` — Merge the bitmap and another bitmap, letting you assign a weight to each channel.

I'll cover each method next.

Drawing from Another Bitmap

The `draw()` method covered in the previous section applies to `BitmapData` objects as well. When you're drawing from other bitmaps, the smoothing parameter may become particularly useful.

Example 36-1 loaded in a bitmap but drew it from the `Loader`, a `DisplayObject`. You can make a slight modification to the code and draw directly from the bitmap, which Example 36-2 demonstrates. Once you do this, you're no longer using `Loader` to display an image — instead, you're using it to decode a compressed image file into usable bitmap data, a trick you've seen in a few places before.

EXAMPLE 36-2 `http://actionscriptbible.com/ch36/ex2`

Capturing Bitmap Data with draw()

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  [SWF(frameRate="10")]
  public class ch36ex2 extends Sprite {
    protected const SIZE:Rectangle = new Rectangle(0, 0, 48, 48);
    protected const SCALE:int = 4;
    protected var TOTALFRAMES:int;
    protected var bmp1:BitmapData; //on the left, no smoothing
    protected var bmp2:BitmapData; //on the right, smoothing
    protected var filmstrip:BitmapData;
    protected var frame:int = 0;
```

```
  public function ch36ex2() {
    var loader:Loader = new Loader();
    //Animation by Derek Yu - www.derekyu.com - used with permission
    loader.load(
      new URLRequest("http://actionscriptbible.com/files/monkey.png"),
      new LoaderContext(true));
    loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    bmp1 = new BitmapData(SIZE.width * SCALE, SIZE.height * SCALE);
    bmp2 = bmp1.clone();
    var bitmap:Bitmap = new Bitmap(bmp1);
    addChild(bitmap);
    bitmap = new Bitmap(bmp2);
    addChild(bitmap);
    bitmap.x = bmp1.width + 10;
  }
  protected function onLoad(event:Event):void {
    filmstrip = Bitmap(LoaderInfo(event.target).content).bitmapData;
    TOTALFRAMES = filmstrip.width / SIZE.width;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  protected function onEnterFrame(event:Event):void {
    frame = ++frame % TOTALFRAMES;
    //clear the bitmaps
    bmp1.fillRect(bmp1.rect, 0);
    bmp2.fillRect(bmp2.rect, 0);
    //shift the filmstrip left to put the correct frame in place; scale up
    var transform:Matrix = new Matrix();
    transform.translate(-frame * SIZE.width, 0);
    transform.scale(SCALE, SCALE);
    //omit sourceRect, let bitmap's own size clip the output
    bmp1.draw(filmstrip, transform, null, null, null, false);
    bmp2.draw(filmstrip, transform, null, null, null, true);
  }
  }
}
```

In Example 36-2, you animate two bitmaps instead of one, and you animate them from a BitmapData that you extract from the Loader (which loads in the image as a Bitmap). Both bitmaps are scaled up, again using a transformation matrix; the one on the left renders without smoothing using the nearest neighbor method to scale up; the one on the right uses interpolation to smooth out the resulting bitmap. Again, don't sweat the transformation matrix, though it's clear what it's doing.

If you run this example, you'll immediately realize that the technique of using a bitmap to scale up images is perfect for retro-styled games, specifically when smoothing is turned off.

Cloning a BitmapData Object

You can create a perfect copy of a bitmap using the clone() method. The method requires no parameters, returning a new BitmapData object:

```
var cloneBitmapData:BitmapData = originalBitmapData.clone();
```

Cloning `BitmapData` objects is useful when you want to create several independent versions of a bitmap. It's not necessary to create clones to display the same bitmap data several times because you can associate one `BitmapData` object with many `Bitmap` objects. However, clones of `BitmapData` objects are useful because changes made to the clones don't affect the original.

In Example 36-2, you used `clone()` in a simple fashion to save a few keystrokes; copying the just-initialized `bmp1` into `bmp2` sets its size to be identical, which is all you're after.

Copying Pixels

You can use the `copyPixels()` method to copy a rectangular region from another bitmap. From a practical perspective, `copyPixels()` is similar to `draw()`. For copying a specific area out of a source bitmap, like you've been doing in the past few examples, it is much easier. The parameters of `copyPixels()` are

- `sourceBitmapData` — The bitmap from which to copy a region.
- `sourceRect` — The region of the source bitmap to copy, as a `Rectangle`.
- `destPoint` — Where the copied pixels should appear in the destination bitmap, as a `Point` defining the top-left corner.
- `alphaBitmapData` — Optionally, a separate bitmap to use as an alpha channel. By default, the alpha channel of the source `BitmapData` object is used. Use this if you have sprites in an opaque image, with a separate image defining a mask.
- `alphaPoint` — Optionally, the origin of the rectangular area to be copied out of the mask bitmap (`alphaBitmapData`), if it doesn't have the same layout as the image data.
- `mergeAlpha` — A Boolean value indicating whether or not to merge the alpha channels of both the source bitmap and `alphaBitmapData`. If `false`, only the `alphaBitmapData`'s alpha channel is used. If true, both alpha channels are blended. The parameter is optional.

In Example 36-3, you'll use the `copyPixels()` method to further simplify the use of a sprite sheet.

EXAMPLE 36-3 `http://actionscriptbible.com/ch36/ex3`

Copying Bitmap Data with copyPixels()

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  [SWF(frameRate="10")]
  public class ch36ex3 extends Sprite {
    protected const SIZE:Rectangle = new Rectangle(0, 0, 48, 48);
    protected var TOTALFRAMES:int;
    protected var bmp:BitmapData;
    protected var filmstrip:BitmapData;
    protected var frame:int = 0;
    protected var sourceRect:Rectangle = SIZE.clone();
    public function ch36ex3() {
      var loader:Loader = new Loader();
      //Animation by Derek Yu – www.derekyu.com – used with permission
```

```
    loader.load(
      new URLRequest("http://actionscriptbible.com/files/monkey.png"),
      new LoaderContext(true));
    loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    bmp = new BitmapData(SIZE.width, SIZE.height);
    var bitmap:Bitmap = new Bitmap(bmp, PixelSnapping.ALWAYS, false);
    bitmap.scaleX = bitmap.scaleY = 4;
    addChild(bitmap);
  }
  protected function onLoad(event:Event):void {
    filmstrip = Bitmap(LoaderInfo(event.target).content).bitmapData;
    TOTALFRAMES = filmstrip.width / SIZE.width;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  protected function onEnterFrame(event:Event):void {
    frame = ++frame % TOTALFRAMES;
    sourceRect.x = frame * SIZE.width;
    bmp.fillRect(bmp.rect, 0);
    bmp.copyPixels(filmstrip, sourceRect, new Point(0,0));
  }
 }
}
```

Here, you see that scaling up the bitmap while drawing isn't the only way to increase the size of the content. You can, instead, simply scale up the `Bitmap` that your bitmap data is displaying in using `DisplayObject` methods, setting the smoothing option on the `Bitmap` instead.

You'll notice that instead of translating and then drawing pixels, `copyPixels()` just lets you specify where the pixels come from and where they go. This makes the example much simpler, but you also have fewer options than with `draw()`, and of course it can only be used to copy regions of a bitmap from another `BitmapData`, not from any `DisplayObject` like `draw()`.

Copying Channels

The `copyChannel()` method lets you copy a portion of a bitmap like `copyPixels()`, but selecting a single channel (red, green, blue, or alpha). In fact, you can even assign pixels from one channel to a different channel using this method. The parameters for `copyChannel()` are as follows:

- `sourceBitmapData` — The bitmap from which to copy a region.
- `sourceRect` — The region of the source bitmap to copy, as a `Rectangle`.
- `destPoint` — Where the copied pixels should appear in the destination bitmap, as a `Point` defining the top-left corner.
- `sourceChannel` — Which color channel to extract from the source bitmap; can be `BitmapDataChannel.ALPHA`, `BitmapDataChannel.RED`, `BitmapDataChannel.GREEN`, or `BitmapDataChannel.BLUE`.
- `destChannel` — Which color channel to place the copied region into (using the same options as before).

In Example 36-4, you'll draw a white circle on a black background, using the drawing API covered in Chapter 35. After capturing the composition as a bitmap, you split it into its three color components,

spreading them out based on the position of the mouse, and clearly showing additive color at work. (The intersection of all three will be white.) You'll see that the ADD blend mode is not used; the three channels are set individually, and the color channels always mix together additively.

EXAMPLE 36-4 http://actionscriptbible.com/ch36/ex4

Copying Color Channels of Bitmap Data

```
package {
  import flash.display.*;
  import flash.events.MouseEvent;
  import flash.geom.Point;
  [SWF(backgroundColor="#000000")]
  public class ch36ex4 extends Sprite {
    protected var center:Point;
    protected var sourceBmp:BitmapData;
    protected var destBmp:BitmapData;
    protected var splitChannelsBitmap:Bitmap;
    public function ch36ex4() {
      //draw a white circle in the center
      center = new Point(stage.stageWidth/2, stage.stageHeight/2);
      var source:Shape = new Shape();
      source.x = center.x; source.y = center.y;
      addChild(source);
      source.graphics.beginFill(0xffffff, 1);
      source.graphics.drawCircle(0, 0, 100);
      //capture a bitmap of the white circle in sourceBmp
      sourceBmp = new BitmapData(stage.stageWidth, stage.stageHeight, false, 0);
      destBmp = sourceBmp.clone();
      sourceBmp.draw(stage);
      //create a holder for destBmp
      splitChannelsBitmap = new Bitmap(destBmp);
      addChild(splitChannelsBitmap);
      stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    }
    protected function onMouseMove(event:MouseEvent):void {
      //hold mouse button to show source graphic
      splitChannelsBitmap.visible = !event.buttonDown;
      var dist:Number = Point.distance(new Point(stage.mouseX, stage.mouseY),
        center) / 2;
      //copy the white circle, offseting RGB channels based on mouse location
      destBmp.copyChannel(sourceBmp, sourceBmp.rect, new Point(0, -dist),
        BitmapDataChannel.RED, BitmapDataChannel.RED);
      destBmp.copyChannel(sourceBmp, sourceBmp.rect, new Point(-dist, dist),
        BitmapDataChannel.GREEN, BitmapDataChannel.GREEN);
      destBmp.copyChannel(sourceBmp, sourceBmp.rect, new Point(dist, dist),
        BitmapDataChannel.BLUE, BitmapDataChannel.BLUE);
      event.updateAfterEvent();
    }
  }
}
```

You'll see that `copyChannel()` works much like `copyPixels()`. You also used the `rect` property of the `BitmapData` object to select the entire area of the bitmap for copying.

Merging BitmapData Images

Use the `merge()` method to blend data from two bitmaps. With this method, you can create a custom mix of the ARGB channels between the two images. I don't use this method often, but you can use it to quickly fade between two images. The method requires these parameters:

- `sourceBitmapData` — The bitmap from which to copy a region.
- `sourceRect` — The region of the source bitmap to copy, as a `Rectangle`.
- `destPoint` — Where the copied pixels should appear in the destination bitmap, as a `Point` defining the top-left corner.
- `redMultiplier` — The weight the source image's red channel will be given, from 0–256. The destination image's red channel will be weighted by the remaining amount (256 – `redMultiplier`). So 128 or 0x80 provides an even mix.
- `greenMultiplier` — The mix for the green channel; see `redMultiplier`.
- `blueMultiplier` — The mix for the blue channel; see `redMultiplier`.
- `alphaMultiplier` — The mix for the alpha channel; see `redMultiplier`.

In Example 36-5, you take two small tiles and blend them gradually, producing a grid of gradations between the two.

EXAMPLE 36-5 `http://actionscriptbible.com/ch36/ex5`

Merging Bitmap Data

```
package {
  import flash.display.*;
  import flash.geom.*;
  import flash.events.Event;
  import flash.net.URLRequest;
  import flash.system.ApplicationDomain;
  import flash.system.LoaderContext;
  public class ch36ex5 extends Sprite {
    public function ch36ex5() {
      var loader:Loader = new Loader();
      loader.load(
        new URLRequest("http://actionscriptbible.com/files/bitmaptiles.swf"),
        new LoaderContext(true));
      loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    }
    protected function onLoad(event:Event):void {
      //extract A and B bitmaps from library SWF
      var L:ApplicationDomain = LoaderInfo(event.target).applicationDomain;
      var tileA:BitmapData = new (Class(L.getDefinition("assets.TileA")))(0,0);
      var tileB:BitmapData = new (Class(L.getDefinition("assets.TileB")))(0,0);

      var steps:int = 60;
      var size:Rectangle = tileA.rect;
```

continued

EXAMPLE 36-5 *(continued)*

```
     var wrapAt:Number = stage.stageWidth;
     var destination:Point = new Point(), origin:Point = new Point();
     var bmp:BitmapData = new BitmapData(stage.stageWidth, stage.stageHeight);
     var tempBmp:BitmapData = new BitmapData(size.width, size.height);
     for (var i:int = 0; i < steps; i++) {
       //clear temp bitmap
       tempBmp.fillRect(size, 0);
       //temp bitmap = A
       tempBmp.copyPixels(tileA, tileA.rect, origin);
       var blend:int = i / steps * 256;
       //blend temp (contains A) and B into temp
       tempBmp.merge(tileB, tileB.rect, origin, blend, blend, blend, blend);
       //copy temp (A-B blend) into next position in grid
       bmp.copyPixels(tempBmp, tempBmp.rect, destination);

       destination.x += size.width;
       if (destination.x + size.width > wrapAt) {
         destination.y += size.height;
         destination.x = 0;
       }
     }
     var bitmap:Bitmap = new Bitmap(bmp);
     addChild(bitmap);
   }
  }
}
```

The tiles here are stored in a SWF as `BitmapData` objects with class linkages. Once you've loaded the SWF and instantiated the bitmaps from its library, a loop generates a grid full of different blends of tile A and tile B into a large bitmap that covers the stage. Because `blend()` operates on the object you call it on and outputs to the object you call it on, you can't blend A and B into the large bitmap. Instead, a temporary bitmap `tmpBmp` is used, first receiving tile A, then blending in the appropriate amount of tile B (storing the result back into `tmpBmp`), and finally copying the blended image into the next slot in a grid.

Pixel-Level Access

For the most direct access to the screen imaginable, you can read and write pixels individually. So far, you've been using loaded images or the drawing API to create bitmaps; with per-pixel access, you can draw directly to the bitmap. Again, this opens the door wide open for any kind of graphics you can program, like custom software renderers, algorithmic art, and optimized rasterizers.

Caution

If you can do something natively in Flash Player, like draw text or curves, it's typically faster to use the native methods to draw and then capture them in a bitmap with `draw()` than to draw them in a bitmap pixel-by-pixel. However, this is a general rule, and there are exceptions. Most often, developers save per-pixel drawing for effects that simply can't be generated with the display list. ∎

There are several methods for accessing the raw bitmap data. Some methods exist to get and set a single pixel, whereas others retrieve and set whole ranges of pixels at once. Because drawing to a bitmap usually entails setting many pixels, often thousands at a time, the methods that copy more bitmap data with a single call are much more efficient.

Accessing Single Pixels

Retrieve or set the color of a single pixel with these methods:

- `getPixel(x:int, y:int):uint` — Return the color of the pixel at (x, y), without its alpha component.
- `getPixel32(x:int, y:int):uint` — Return the color of the pixel.
- `setPixel(x:int, y:int, color:uint):void` — Set the color of the pixel, without its alpha component.
- `setPixel32(x:int, y:int, color:uint):void` — Set the color of the pixel, including its alpha component.

The 32-bit methods accept and return 32-bit integers packed as ARGB, such as 0xFF00CED1, an opaque turquoise, while the 24-bit methods accept and return 24-bit integers packed as RGB, such as 0x00CED1, the same color.

When you're drawing any significant number of pixels, use the `lock()` and `unlock()` methods between batches of `setPixel()` calls. These prevent Flash Player from doing extra work repeatedly updating the on-screen representation of the bitmap, when it's just going to be changed again by the next `setPixel()` call.

In Example 36-6, you'll use `setPixel()` to draw alpha-blended circles, in which the color drops off towards the edge of the circle.

EXAMPLE 36-6 | `http://actionscriptbible.com/ch36/ex6`

Drawing Pixel by Pixel

```
package {
  import flash.display.*;
  import flash.geom.*;
  import flash.events.MouseEvent;
  public class ch36ex6 extends Sprite {
    protected const AVG_RADIUS:Number = 30;
    protected var bmp:BitmapData;
    public function ch36ex6() {
      bmp = new BitmapData(stage.stageWidth, stage.stageHeight);
      var bitmap:Bitmap = new Bitmap(bmp);
      addChild(bitmap);
      stage.addEventListener(MouseEvent.CLICK, onClick);
    }
    protected function onClick(event:MouseEvent):void {
      bmp.lock();
      var radius:Number = (Math.random() - 0.5) * AVG_RADIUS + AVG_RADIUS;
      var center:Point = new Point(event.localX, event.localY);
      var bounds:Rectangle = new Rectangle(center.x, center.y, 0, 0);
      bounds.inflate(2*radius, 2*radius);
```

continued

EXAMPLE 36-6   *(continued)*

```
    bounds = bounds.intersection(bmp.rect);
    var p:Point = new Point();
    for (p.y = int(bounds.top); p.y < bounds.bottom; p.y++) {
      for (p.x = int(bounds.left); p.x < bounds.right; p.x++) {
        var dist:Number = Point.distance(p, center);
        if (dist < radius) {
          var alpha:uint = 0xff * (1 - dist / radius);
          bmp.setPixel32(p.x, p.y, alpha << 24);
        }
      }
    }
    bmp.unlock(bounds);
  }
 }
}
```

You'll have to use the implicit definitions of any geometry you'd like to draw. For example, the circle you drew in the example appears when you color every pixel within its radius. If you run the example, you'll notice that while the edges of each circle fade out, overlapping circles don't blend. This should be no surprise, since each click overwrites the pixels with no concern for their prior contents. For an additional challenge, you could blend these easily, either by drawing the circle into a new bitmap and merge()ing them, or by blending the pixels with getPixel32() inside the inner loop.

You'll notice that all of the setPixel() calls are surrounded by a pair of lock() and unlock() calls. Whether you include these calls or not, you'll never see the pixels be set gradually on screen, not simply because they're set too fast for you to catch, but because the screen is only refreshed after all ActionScript 3.0 code in a frame has been executed. The loops in onClick() are run synchronously, and the image appears at the next redraw. But locking and unlocking is still important. Flash Player needs to get the hint from you. Passing the bounds rectangle to the unlock() call in Example 36-6 is another, optional, hint. It tells Flash Player which area of the bitmap has changed; when you tell Flash Player, it saves it the time and effort of finding out. Conveniently, in the example, you've precalculated that value.

Accessing Bitmap Data in Memory

In the beginning of this chapter, you learned that bitmaps are nothing more magical than a huge array of color values — but that the BitmapData class manages this data. So you end up using setPixel() and getPixel() methods to access pixels rather than something simple — and fast — like direct array access.

```
    var bmp:BitmapData = new BitmapData(100, 100);
    bmp[50][50]; //XXX not available
    bmp.getPixel(50, 50); //OK, but cumbersome
```

Rather than using these somewhat cumbersome methods thousands of times to manipulate a whole region of pixels, BitmapData will allow you to get a copy of a region (which can, in fact, be the entire image) in a more comfortable data type.

Using ByteArray

The `getPixels()` and `setPixels()` methods provide access to bitmap data as a `ByteArray`. Each pixel color is stored as a 32-bit `uint`. Pixels are stored in row-major order, that is, how you would read the pixels if they were an English-language newspaper: left-to-right, top-to-bottom. Each of the methods accepts a region as a `Rectangle`, which the pixels will be read from, or written into.

You might want to use a `ByteArray` for bitmap data when importing or exporting image data, from a file or possibly even a network connection. For instance, the screen-sharing application VNC operates by continuously sending compressed image data for small changed regions of the screen. You could use a `Socket`, receive `ByteArrays`, decode them, and copy them into the screen `BitmapData` using `setPixels()`.

A distinct advantage of using `ByteArray` for bitmap operations is that `ByteArray` permits array-style access with square brackets (`[]`). Moreover, because it's a *byte* array, an index in this style returns a single byte. In 32-bit bitmap data, a single byte corresponds to a single channel of a pixel. So you can use array access notation to easily address a single channel of a single pixel, as Example 36-7 shows.

EXAMPLE 36-7 `http://actionscriptbible.com/ch36/ex7`

Accessing Bitmap Data in a ByteArray

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  import flash.utils.ByteArray;
  [SWF(backgroundColor="#000000")]
  public class ch36ex7 extends Sprite {
    public function ch36ex7() {
      var l:Loader = new Loader();
      l.load(
        new URLRequest("http://actionscriptbible.com/files/testpattern.png"),
        new LoaderContext(true)
      );
      l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    }
    protected function onLoad(event:Event):void {
      var src:BitmapData = Bitmap(LoaderInfo(event.target).content).bitmapData;
      //setting the alpha channel doesn't work if the image isn't transparent
      //so you copy the bitmap data into a new transparent BitmapData
      var bmp:BitmapData = new BitmapData(src.width, src.height, true);
      bmp.draw(src);
      var bytes:ByteArray = bmp.getPixels(bmp.rect);
      for (var y:int = 0; y < bmp.height; y+=2) {
        for (var x:int = 0; x < bmp.width; x++) {
          //find the alpha channel of x,y
          bytes[y * bmp.width * 4 + x * 4] = 0x0;
        }
      }
```

continued

EXAMPLE 36-7   *(continued)*

```
      //you must rewind the ByteArray first, for some reason
      bytes.position = 0;
      bmp.setPixels(bmp.rect, bytes);
      var bitmap:Bitmap = new Bitmap(bmp);
      addChild(bitmap);
    }
  }
}
```

In Example 36-7, you use a `ByteArray` to darken out every other row of pixels like an old-school TV monitor, by zeroing out each pixel's alpha channel. Using a `ByteArray`, you can do so with per-byte array access. Because each pixel takes up four bytes and the pixels are stored in row-major order, channel number `C` of pixel `x`, `y` is stored at

```
      y * (width of a row) * (number of channels) + x * (number of channels) + C
      4 * y * width + 4 * x + C
```

Two complications arise in the example. The alpha channel is ignored when the bitmap does not have its `transparent` flag set. And the `setPixels()` method starts reading pixels at the cursor position of the `ByteArray`, so you'll need to rewind it to the correct position if you've done any writing.

Using Vector

In Flash Player 10 and later, `BitmapData` lets you access regions of pixels as `Vectors` of `uints`. Besides being awfully convenient, this method is hands-down the fastest way to achieve direct access to bitmap data. The `getVector()` and `setVector()` methods provide `Vector` access to bitmap data, also with a region `Rectangle`. Other than the difference in data type, these work exactly like `getPixels()` and `setPixels()`.

Version

FP10. The `getVector()` and `setVector()` methods of `BitmapData`, like `Vector` itself, are available only in Flash Player 10 and later. ■

Note

Programmer Nathan Rixham shares his speed comparison of different per-pixel bitmap access methods in a nice article at `http://bit.ly/direct-bitmap-access-compared`. ■

Again, the pixels are stored in row-major order and packed into 32-bit `uints` in ARGB order. `Vectors`, you'll recall, also allow array-style access, but because the parameterized type of these `Vectors` is `uint`, each index addresses a single pixel.

Use a `Vector` to access bitmap data when you're writing your own renderers and rasterizer techniques and when your application will be run in Flash Player 10 or later.

Note that when you use a `Vector` or a `ByteArray`, it's not necessary to `lock()` the `BitmapData`, because you're only using a single call to write a whole set of pixels.

Working with Colors

Bitmaps provide some useful techniques for analyzing and manipulating colors within an image.

Filling with a Solid Color

Fill a rectangular region of a bitmap with a solid color using the `fillRect()` method. The method requires two parameters: a `Rectangle` defining the region to fill, and the color. You've already seen this rather simple method in many of the examples, usually used to clear the entire bitmap.

```
var bmp:BitmapData = new BitmapData(300, 300);
//draw a solid blue rectangle in the middle
bmp.fillRect(new Rectangle(100, 100, 100, 100), 0xff0000ff);
//clear the whole thing (write transparent black)
bmp.fillRect(bmp.rect, 0);
```

Replacing a Color with a Flood Fill

Flood fills replace an area of contiguous color with another color, essentially changing the color of any shape with a solid fill. Without being told the boundaries of the shape, the fill starts at an origin point and floods outward in every direction until the color changes — even a little bit. You're probably familiar with this tool from image editing programs.

Apply flood fills to bitmaps using the `floodFill()` method, which takes the start position of the fill and the new color, as Example 36-8 shows. Unlike the flood fill tool in an image editing program, you really can't change the threshold: the flood fill extends only to areas of *precisely* the same color.

EXAMPLE 36-8 `http://actionscriptbible.com/ch36/ex8`

Flood Fills

```
package {
  import flash.display.*;
  import flash.events.MouseEvent;
  import flash.geom.Rectangle;
  public class ch36ex8 extends Sprite {
    protected var bmp:BitmapData;
    public function ch36ex8() {
      bmp = new BitmapData(200, 200, false);
      bmp.fillRect(new Rectangle(0, 0, 100, 100), 0xffff0000);
      bmp.fillRect(new Rectangle(100, 0, 100, 100), 0xff00ff00);
      bmp.fillRect(new Rectangle(0, 100, 100, 100), 0xff0000ff);
      bmp.fillRect(new Rectangle(100, 100, 100, 100), 0xffffff00);
      var bitmap:Bitmap = new Bitmap(bmp);
      addChild(bitmap);
      stage.addEventListener(MouseEvent.CLICK, onClick);
    }
```

continued

EXAMPLE 36-8 *(continued)*

```
    private function onClick(event:MouseEvent):void {
      bmp.floodFill(event.localX, event.localY, Math.random() * 0xffffff);
    }
  }
}
```

Color Transforms

Recall from Chapter 34 that you can change the appearance of any display object by modifying its color transform. `BitmapData` is not a `DisplayObject`, but you can still apply a color transform to it. Unlike display objects, which retain their transformations and are continually affected by them, bitmaps are static, so the `colorTransform()` method permanently modifies the color of every pixel. The `colorTransform()` method requires two parameters: a `Rectangle` object defining the region to which to apply the color transform, and a `ColorTransform` object to use.

Color transforms are really useful when you want to use a bitmap as a frame buffer, drawing new frames on top of older frames. To create a great motion trail effect with little effort, just dim the previous contents of the frame before drawing a new frame, as shown in Example 36-9.

EXAMPLE 36-9 http://actionscriptbible.com/ch36/ex9

Applying Color Transforms

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.geom.*;
  [SWF(backgroundColor="#000000",frameRate="60")]
  public class ch36ex9 extends Sprite {
    protected const MAXSPEED:Number = 8;
    protected var ball:Shape;
    protected var holder:Sprite;
    protected var bmp:BitmapData;
    protected var velocity:Point;
    public function ch36ex9() {
      bmp = new BitmapData(stage.stageWidth, stage.stageHeight);
      var bitmap:Bitmap = new Bitmap(bmp);
      addChild(bitmap);
      holder = new Sprite();
      addChild(holder);
      ball = new Shape();
      ball.graphics.lineStyle(0, 0xff0000);
      ball.graphics.beginFill(0xffffff, 0.1);
      ball.graphics.drawCircle(0, 0, 30);
      holder.addChild(ball);
      stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
      stage.addEventListener(MouseEvent.CLICK, onMouseClick);
      velocity = new Point(1, 1);
    }
```

```
    protected function onEnterFrame(event:Event):void {
      ball.x += velocity.x;
      ball.y += velocity.y;
      ball.rotation += velocity.length / 5;
      if (ball.x >= bmp.width || ball.x <= 0) velocity.x *= -1;
      if (ball.y >= bmp.height || ball.y <= 0) velocity.y *= -1;
      bmp.colorTransform(bmp.rect, new ColorTransform(1, 1, 1, 0.9));
      bmp.draw(holder);
    }
    protected function onMouseClick(event:MouseEvent):void {
      ball.x = stage.mouseX;
      ball.y = stage.mouseY;
      velocity.x = Math.random() * MAXSPEED * 2 - MAXSPEED;
      velocity.y = Math.random() * MAXSPEED * 2 - MAXSPEED;
    }
  }
}
```

In the example, you implement a simple motion trail by drawing every frame into the same bitmap, while fading out the contents continually with a `ColorTransform`.

Retrieving a Histogram

A histogram shows the distribution of colors in an image. In Flash Player, a bitmap's histogram is provided for each of the four channels individually. Each channel's histogram is a plot of the luminance value (between 0 and 255) and its frequency — the number of pixels in the image that have that value for the specified channel.

You can use a histogram to quickly get a glance of the dominant colors in the image and their luminances. By summing up the contributions from the RGB channels, you can look at the distribution of light levels in an image, too. This information is useful in computer vision algorithms, often as a first-pass analysis. For instance, when attempting to track a color marker, you can look for a significant amount of the target color in the image; if it doesn't appear, you can save processor time by not even attempting to locate the marker.

When using a histogram to do image analysis, you'll likely want to smooth out the data to minimize impact of noise and artifacts. And you'll usually be looking for local and global minima and maxima to identify dominant colors or brightnesses. In object recognition algorithms, you may also want to retrieve the histogram of a subset of the image to try to zero in on the location of the object.

Retrieve the histogram of an image, or a region of interest (ROI) within an image, by calling the `histogram()` method. The method optionally takes a `Rectangle` defining the region of interest. It returns a `Vector` of `Vectors` of `Numbers` — in other words, four histograms, one for each channel, each with 256 frequencies, indicating the number of pixels found in the bitmap with the corresponding brightness.

Version

FP10. You can use the `histogram()` method in Flash Player 10 and later. In Flash Player 9, you can generate a histogram manually with pixel-by-pixel tallying of the entire image. ■

In Example 36-10, you'll draw a live histogram over the webcam's image, in much the same way that you'd write a spectrum visualizer for sound.

EXAMPLE 36-10 http://actionscriptbible.com/ch36/ex10

Getting an Image's Histogram

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.media.Camera;
  import flash.media.Video;
  [SWF(width="500",height="500",frameRate="20")]
  public class ch36ex10 extends Sprite {
    protected const COLORS:Vector.<uint> = new <uint>[0xff0000, 0xff00, 0xff];
    protected const SCALE:Number = 200 / 256;
    protected var bmp:BitmapData;
    protected var video:Video;
    protected var hstR:Shape, hstG:Shape, hstB:Shape;
    protected var allHstShapes:Vector.<Shape>;
    public function ch36ex10() {
      bmp = new BitmapData(stage.stageWidth, stage.stageHeight);
      video = new Video(stage.stageWidth, stage.stageHeight);
      video.attachCamera(Camera.getCamera());
      addChild(video);
      stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
      var hstbox:Sprite = new Sprite();
      addChild(hstbox);
      hstbox.x = stage.stageWidth - 270 - 10;
      hstbox.y = 10;
      hstbox.graphics.lineStyle(0, 0xffffff);
      hstbox.graphics.beginFill(0, 0.5);
      hstbox.graphics.drawRect(0, 0, 270, 150);
      hstR = new Shape(); hstG = new Shape(); hstB = new Shape();
      allHstShapes = new <Shape>[hstR, hstG, hstB];
      for each (var hstShape:Shape in allHstShapes) {
        hstbox.addChild(hstShape);
        hstShape.y = hstbox.height - 10;
        hstShape.blendMode = BlendMode.ADD;
        hstShape.rotationY = 36;
        hstShape.scaleY = -1;
      }
      hstR.x = 10; hstG.x = 30; hstB.x = 50;
    }
    protected function onEnterFrame(event:Event):void {
      bmp.draw(video);
      var allHstData:Vector.<Vector.<Number>> = bmp.histogram(bmp.rect);
      var PIXELCOUNT:int = bmp.width*bmp.height;
      for (var i:int = 0; i < allHstShapes.length; i++) {
        var hstData:Vector.<Number> = allHstData[i];
        var g:Graphics = allHstShapes[i].graphics;
        g.clear();
```

```
      g.beginFill(COLORS[i]);
      for (var x:int = 0; x < hstData.length; x++) {
        g.drawRect(x*SCALE, 0, SCALE, hstData[x]/PIXELCOUNT * 1500*SCALE);
      }
    }
  }
}
}
```

Although `histogram()` generates four histograms, the fourth, alpha channel, is ignored in this example because the webcam feed is completely opaque.

Replacing Colors with Threshold

Another one of the simplest and most useful tools for image processing and computer vision is the threshold. Computer programs, with their binary logic, have a hard time making decisions on images with their 16.7 million possible values each pixel. Running a threshold operation reduces an image to a black-and-white, yes-or-no answer for every pixel. The threshold usually asks the question: is this pixel's color value greater than a certain target? Actually, a threshold can perform any numerical comparison on pixels: greater than, less than, equal, not equal, and so on. After reducing an image to binary, it's much simpler to analyze and make decisions about.

Computer vision programs frequently use thresholds to determine which areas of an image are noteworthy, and they focus on them. For instance, you can threshold an image to try to determine the foreground. In fact, you can use the information a histogram provides about an image to decide how to threshold it for a better result.

The `threshold()` method performs a threshold operation. Like `copyPixels()`, it has an input and output bitmap; often you want to use the thresholded image as additional information about the image, without replacing and losing the original image data. The method accepts the following parameters:

- `sourceBitmapData` — The bitmap from which to copy a region.
- `sourceRect` — The region of the source bitmap to copy, as a `Rectangle`.
- `destPoint` — Where the copied pixels should appear in the destination bitmap, as a `Point` defining the top-left corner.
- `operation` — The comparison to perform on each pixel. One of the following as a `String`: <, >, <=, >=, !=, or ==.
- `threshold` — The color to compare against.
- `color` — Pixels for which the threshold test results in `true` will be set to this color.
- `mask` — If provided, input color values will be masked with this value before comparison; for example, `color & mask == threshold & mask`. Optional.
- `copySource` — If `true`, the source pixel's color is used when the threshold test fails. Optional; defaults to `false`.

755

In Example 36-11, you black out all pixels of a webcam feed that are brighter than the dominant color. This should help isolate objects from a solid, bright background or light source. However, if the image is predominantly dark, this might black out the whole image. A better approach might be to use the brightest *local* maximum to threshold rather than the global maximum. You can experiment with different approaches.

EXAMPLE 36-11 http://actionscriptbible.com/ch36/ex11

Applying a Threshold

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.Point;
  import flash.media.Camera;
  import flash.media.Video;
  public class ch36ex11 extends Sprite {
    protected var bmp:BitmapData;
    protected var thresholdBmp:BitmapData;
    protected var video:Video;
    public function ch36ex11() {
      bmp = new BitmapData(400, 300);
      addChild(new Bitmap(bmp));
      video = new Video(400, 300);
      video.attachCamera(Camera.getCamera());
      stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      bmp.draw(video);
      var hst:Vector.<Vector.<Number>> = bmp.histogram(bmp.rect);
      var maxValue:Vector.<int> = new Vector.<int>(3, true);
      var maxFrequency:Vector.<Number> = new Vector.<Number>(3, true);
      for (var channel:int = 0; channel < 3; channel++) {
        for (var i:int = 0; i < 256; i++) {
          var value:Number = hst[channel][i];
          if (value > maxFrequency[channel]) {
            maxFrequency[channel] = value;
            maxValue[channel] = i;
          }
        }
      }
      var threshold:uint = maxValue[0] << 16 | maxValue[1] << 8 | maxValue[2];
      bmp.threshold(bmp, bmp.rect, new Point(),
        ">", threshold, 0xff000000, 0x00ffffff, true);
    }
  }
}
```

Remapping Colors with Palette Mapping

Color transforms are great for changing relative levels of colors in an image, but their simple linear formula restrains their effects. When you want to get really freaky with color, you can define a completely arbitrary mapping between colors. Yes, you can actually pick a new value for each one of the 16.7 million colors in an image (4.2 billion when you count alpha). You're probably sick of hearing this, but this gives you limitless control over an image's colors.

You can use palette mapping to more effectively replace single colors or whole ranges of colors. You can use them to easily shift hues or change the response curves (gamma) of an image. In classic computer games, images were often stored as indexed colors, and often with 256 or fewer colors. Then palette mapping was a cheap and trippy special effect, which you can re-create in Flash Player.

Remap the palette of a bitmap using `paletteMap()`. Like `copyPixels()`, it can copy from a source to a destination bitmap, placing a particular region in a specific destination. Of course, you can use the entire bitmap and copy it into itself if you wish. The `paletteMap()` method accepts the following parameters:

- `sourceBitmapData` — The bitmap from which to copy a region.
- `sourceRect` — The region of the source bitmap to copy, as a `Rectangle`.
- `destPoint` — Where the copied pixels should appear in the destination bitmap, as a `Point` defining the top-left corner.
- `redArray` — An array defining the mapping for the red channel.
- `greenArray` — An array defining the mapping for the green channel.
- `blueArray` — An array defining the mapping for the blue channel.
- `alphaArray` — An array defining the mapping for the alpha channel.

The four arrays define the mapping. Rather than specifying a mapping from 32-bit color A to 32-bit color B, you define a mapping between the individual channels, although you can easily construct those color-to-color mappings in code. The index of each array (between 0 and 255) represents the color component value; the value at the index, an integer between 0 and 255, defines what the new color component will be. These array parameters are optional. If they are omitted or `null`, the corresponding color channel will not be changed.

In Example 36-12, you construct a palette mapping that modifies the gamma of a video as you move your mouse. You can change gamma depending on the platform as a simple form of color correction, because Mac platforms and Windows platforms typically drive their displays at different gamma values (although this is no guarantee that the user's display isn't set differently). As a better alternative, you should use `stage.colorCorrection` if available (in Flash Player 10 and later). See Chapter 41, "Globalization, Accessibility, and Color Correction," to find out how.

EXAMPLE 36-12 `http://actionscriptbible.com/ch36/ex12`

Adjusting Gamma with Palette Mapping

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

EXAMPLE 36-12   *(continued)*

```
import flash.geom.Point;
import flash.media.Video;
import flash.net.*;
import flash.text.*;
public class ch36ex12 extends Sprite {
  protected var RLUT:Array, GLUT:Array, BLUT:Array;
  protected var video:Video;
  protected var bmp:BitmapData;
  protected var tf:TextField;
  public function ch36ex12() {
    RLUT = new Array(256);
    GLUT = new Array(256);
    BLUT = new Array(256);
    var nc:NetConnection = new NetConnection();
    nc.connect(null);
    var ns:NetStream = new NetStream(nc);
    ns.play("http://actionscriptbible.com/files/bullettrain.mp4");
    ns.client = new Object();
    video = new Video();
    video.attachNetStream(ns);
    video.scaleX = video.scaleY = 0.5;
    bmp = new BitmapData(320, 240);
    addChild(new Bitmap(bmp));
    video.addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    tf = new TextField(); tf.width = 26; tf.height = 14; tf.x = tf.y = 5;
    tf.backgroundColor = 0; tf.background = true;
    tf.defaultTextFormat = new TextFormat("_typewriter", 10, 0xffffff);
    addChild(tf);
  }
  protected function onEnterFrame(event:Event):void {
    bmp.draw(video);
    bmp.paletteMap(bmp, bmp.rect, new Point(), RLUT, GLUT, BLUT);
  }
  protected function onMouseMove(event:MouseEvent):void {
    var newGamma:Number = (1 - stage.mouseY / stage.stageHeight) * 2.5;
    tf.text = newGamma.toFixed(1);
    for (var v:int = 0; v < 256; v++) {
      BLUT[v] = int(Math.pow(v / 256.0, newGamma) * 256);
      GLUT[v] = BLUT[v] << 8;
      RLUT[v] = GLUT[v] << 8;
    }
  }
}
}
```

You might also consider modifying the response curve of an image at real time to simulate changes in exposure, such as if the character steps outside into a bright scene and his eyes adjust.

Detecting Areas of a Solid Color

Detecting the extents of a color is a powerful image analysis technique, especially when the colors of an image have been simplified a bit (as this method, like `floodFill()` discussed earlier, has zero tolerance). For example, once you have determined that a solid-colored feature you would like to recognize is within some region of interest, you can find the smallest bounding region that contains the color and determine its position from this (perhaps with temporal interpolation, taking its prior locations into account).

To find out the smallest region that encloses the solid-colored area of a bitmap, use the `getColorBoundsRect()` method. The method has some things in common with `threshold()` in the way it compares pixels of a bitmap to a target color. It accepts the following parameters:

- `mask` — Causes pixel values to be masked prior to comparison. The comparison `pixel & mask == color & mask` is used. Particularly useful if you're only interested in the alpha channel (use 0xFF000000), or not interested in the alpha channel (use 0x00FFFFFF). Use 0xFFFFFFFF to consider all channels.

- `color` — A color to detect the bounds of.

- `findColor` — If `true`, returns the area containing the color; otherwise, returns the area in which the color is not found. Optional; defaults to `true`.

The method returns a `Rectangle` object enclosing the region that contains the color.

In Example 36-13, left-aligned text is wrapped around an image by detecting, for each line, where that line overlaps the image, and then within that overlap, how much of the image is transparent (or solid white, in an opaque image).

EXAMPLE 36-13   `http://actionscriptbible.com/ch36/ex13`

Detecting Solid Color

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.net.*;
  import flash.text.*;
  public class ch36ex13 extends Sprite {
    protected var tf:TextField;
    protected var img:Loader;
    public function ch36ex13() {
      tf = new TextField();
      tf.width = stage.stageWidth * 0.75;
      tf.height = stage.stageHeight - 4;
      tf.x = 4; tf.y = 4;
      tf.defaultTextFormat = new TextFormat("Garamond", 12, 0x303030);
      tf.multiline = tf.wordWrap = true;
      addChild(tf);
      var u:URLLoader = new URLLoader(
        new URLRequest("http://actionscriptbible.com/files/alice-ch8.txt"));
      u.addEventListener(Event.COMPLETE, onTextLoad);
    }
```

continued

EXAMPLE 36-13   *(continued)*

```
    protected function onTextLoad(event:Event):void {
      tf.text = URLLoader(event.target).data;
      img = new Loader();
      addChildAt(img, 0);
      img.load(
        new URLRequest("http://actionscriptbible.com/files/alice-8.gif"));
      img.contentLoaderInfo.addEventListener(Event.COMPLETE, onImageLoad);
    }
    protected function onImageLoad(event:Event):void {
      img.x = stage.stageWidth - img.width + 30;
      img.y = 30;
      WrapTextUtility.wrapText(tf, img, 2, true);
    }
  }
}

import flash.geom.*;
import flash.display.*;
import flash.text.*;
class WrapTextUtility
{
  private static const KILL_LIMIT:int = 100;
  private static const GUTTER:int = 2;
  private static const NEWLINE:String = "\n";
  private static const WHITESPACE:RegExp = /[\s\-\_]/;
  public static function wrapText(tf:TextField, edge:DisplayObject,
                                  paddingPx:int = 6,
                                  transparentBackground:Boolean = false):void {
    if (!tf.hitTestObject(edge)) return;
    var edgeOffset:Point = edge.localToGlobal(new Point()).subtract(
      tf.localToGlobal(new Point()));
    var lineY:Number = GUTTER;
    var bmpSlice:BitmapData = new BitmapData(edge.width, edge.height, true, 0);
    var i:int = 0;
    do {
      try {
        var lineMetrics:TextLineMetrics = tf.getLineMetrics(i);
      } catch (err:RangeError) {
        break;
      }
      var lineBaseline:Number = lineY + lineMetrics.ascent;
      bmpSlice.fillRect(bmpSlice.rect, 0x00000000);
      var clipRect:Rectangle = new Rectangle(0, lineY - edgeOffset.y,
        edge.width, lineMetrics.ascent + lineMetrics.descent);
      lineY += lineMetrics.height;
      if (clipRect.width <= 0 || clipRect.height <= 0)
        continue;
      if (clipRect.y >= edge.height || clipRect.y + clipRect.height <= 0)
        continue;
      bmpSlice.draw(edge, null, null, null, clipRect);
      var colorRect:Rectangle = bmpSlice.getColorBoundsRect(
```

```
      (transparentBackground? 0xFF000000 : 0xFFFFFFFF), 0x00000000, false);
    if (colorRect == null || colorRect.width <= 0 || colorRect.height <= 0)
      continue;
    var wrapChar:int = tf.getCharIndexAtPoint(
      colorRect.x + edgeOffset.x - paddingPx, lineBaseline);
    var firstCharInLine:int = tf.getLineOffset(i);
    var allText:String = tf.text;
    if (wrapChar <= 0) continue;
    while (allText.charAt(wrapChar).match(WHITESPACE) == null &&
      wrapChar > firstCharInLine) --wrapChar;
    if (wrapChar <= firstCharInLine) {
      tf.text = allText.slice(0, firstCharInLine)
        + NEWLINE
        + allText.slice(firstCharInLine);
    } else {
      tf.text = allText.slice(0, wrapChar)
        + NEWLINE
        + allText.slice(wrapChar + 1);
    }
  } while (++i < KILL_LIMIT);
  bmpSlice.dispose();
  }
}
```

Once the line's overlap with the image is detected, you create a small bitmap slice (`bmpSlice`) that represents that overlap. Then you use `getColorBoundsRect()` to see which part of it can have text drawn on top (if it's transparent or white). Then you manually wrap the text to the right edge of that rectangle.

Bitmap Effects

With the use of some final methods by the `BitmapData` class, you can generate interesting effects. Some of these, such as Perlin noise, can be generated in a bitmap and then used in a wide variety of other contexts.

Applying Filters

Filters, covered in Chapter 37, can be applied to any display object. The `BitmapData` class isn't a `DisplayObject`, but you can apply a filter to it. Much like `colorTransform()`, the filter will have a one-time effect. However, you can certainly apply the filter repeatedly, especially when maintaining the prior contents of the frame buffer as in Example 36-9.

To apply a filter to a `BitmapData` object, use the `applyFilter()` method, which accepts the following parameters:

- `sourceBitmapData` — The bitmap to apply the filter to.
- `sourceRect` — The region of the source bitmap to copy, as a `Rectangle`.
- `destPoint` — Where the copied pixels should appear in the destination bitmap, as a `Point` defining the top-left corner.
- `filter` — A `BitmapFilter` to apply.

761

Scrolling a Bitmap

You can move a bitmap's data around in a simple manner with the `scroll()` method. The same effect can be achieved with `copyPixels()` or `draw()`, but `scroll()` is quick and easy. Rather than using a source bitmap, the `BitmapData` shifts its own contents. The method takes an x and y delta in position. When using `scroll()`, you should have a larger bitmap than is shown on-screen: the source pixels have to come from somewhere as they are shifted.

You can use `scroll()` to create a smoothly scrolling tile map. Tile maps are an old-school game technique to create and store large maps at a low cost. Game artists would create patches of ground that fit together seamlessly and reuse them to build a large map. As the game character walks toward the edge of the screen, new tiles have to be brought on from the edge. You can draw on these tiles with `copyPixels()`, but to generate the frames of animation where the screen is smoothly scrolling halfway between two unit tile sizes, `scroll()` can come in handy. The bitmap loses data as pixels are pushed off the edge, so it comes at a cost: you have to redraw the map after scrolling changes directions.

You can also use `scroll()` for simple frame buffer effects, such as fire that scrolls up and fades out gradually. Each frame, the whole bitmap can shift up, fade out, and perhaps blur.

Using Pixel Dissolves

Although perhaps one of the less useful features of `BitmapData`, you can copy a random set of pixels from a bitmap, usually to evince a "pixel dissolve," a transition between two images, or an image and black, in which pixels progressively disappear in a random fashion. The `pixelDissolve()` method does just this.

Making Noise

Noise typically refers to random, unintentional, and unwanted image data. However, there are many good reasons to intentionally introduce noise. For example, you can use noise to add texture like film grain; and fractal noise is useful for simulating natural motion. There are two methods to create noise in a bitmap.

Random Noise

The `noise()` method applies randomly distributed noise to a bitmap. This kind of noise is akin to film grain or static. The noise is applied to the entire image. The noise at each pixel is a random value conforming to certain parameters. The `noise()` method accepts the following parameters, which are all optional but the first:

- `randomSeed` — A seed for the random number generator. Because the random numbers are pseudorandom, use of the same seed generates the same noise (not that anyone is likely to notice). A convenient random seed that's not likely to be seen twice by the same person is the current time, `(new Date()).getTime()`.

- `low` — The minimum possible value for each channel (ARGB). Optional; defaults to 0.

- `high` — The maximum possible value for each channel. Optional; defaults to 255.

- `channelOptions` — Which channels noise will be added to. Use a combination of the `RED`, `GREEN`, `BLUE`, and `ALPHA` constants of `BitmapDataChannel`, or their values, 1, 2, 4, and 8. Use multiple channels with addition (+) or binary OR (|). Optional, defaults to RGB.

- grayScale — Whether to create grayscale noise. If true, the same random value is used for all channels specified in channelOptions, and the noise appears different levels of gray. Optional; defaults to false.

Example 36-14 uses the noise() method to attempt realistic film grain. Because the per-pixel noise is so sharp, a blur filter is used. This, and the multiply blend mode, help make the noise appear more realistic. Finally, add in a flicker and set the frame rate to 24 to make it totally filmic.

EXAMPLE 36-14 http://actionscriptbible.com/ch36/ex14

Adding Noise

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.filters.BlurFilter;
  import flash.geom.Point;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  [SWF(frameRate="24", backgroundColor="#000000")]
  public class ch36ex14 extends Sprite {
    protected var bmp:BitmapData;
    protected var seed:int;
    protected var image:Loader;
    public function ch36ex14() {
      image = new Loader();
      addChild(image);
      image.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
      image.load(
        new URLRequest("http://actionscriptbible.com/files/caviar.jpg"),
        new LoaderContext(true)
      );
    }
    protected function onLoad(event:Event):void {
      var src:BitmapData = Bitmap(image.content).bitmapData;
      bmp = new BitmapData(src.width, src.height, true);
      var bitmap:Bitmap = new Bitmap(bmp);
      addChild(bitmap);
      bitmap.blendMode = BlendMode.MULTIPLY;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      bmp.fillRect(bmp.rect, 0);
      bmp.noise(++seed, 200, 255, BitmapDataChannel.BLUE, true);
      bmp.applyFilter(bmp, bmp.rect, new Point(), new BlurFilter(4, 4, 2));
      image.alpha = (seed % 2)? 0.95 : 1;
    }
  }
}
```

Notice how you also used a bitmap filter with applyFilter() to soften the otherwise-harsh grain.

763

Perlin Noise

Perlin noise is a kind of fractal noise. Unlike the harsh grit of totally random noise, Perlin noise is flowing, liquid, organic. It's fractal because it contains multiple layers of the same kind of noise, each at a smaller size — called *octaves* — so it's self-similar. Usually, only a few octaves are shown, unlike fractals that iterate toward infinity. Many aspects of the noise are controlled by code, so that you can corral the effect to your own ends. The usefulness of Perlin noise extends beyond creating plasma-like blobs; you can even generate an image with Perlin noise and examine its pixels as a replacement for `Math.random()` in totally nonbitmap code.

It's hard to visualize Perlin noise from a short description; you can see a basic example of it in Figure 36-1. Without much manipulation, you can make Perlin noise approximate wood grain, marble, and clouds. Using randomness it outputs, you can model terrain, wind, fire, and ripples. Those examples are just the most common applications. Use Perlin noise a few times, and you may find yourself realizing it'd be perfect for a completely different application.

FIGURE 36-1

Basic Perlin noise

The `perlinNoise()` method fills a bitmap with Perlin noise. Control the noise generation with the following parameters:

- `baseX` — The width to base the frequency of noise on. Use the width of the bitmap as a standard of measurement. Fit more or less in the bitmap by scaling this factor up and down.

- `baseY` — The height to base the frequency of noise on. See `baseX`.

- `numOctaves` — The number of octaves to render. Typically only a few octaves are rendered. Fewer octaves mean faster render times.

- `randomSeed` — A seed value for the pseudorandom number generator. Use the same seed to repeat the same general appearance. Changes in seed value don't appear continuous.

- `stitch` — Whether the edges of the noise are made to fit together, should the noise be tiled.

- `fractalNoise` — Whether the style of noise should be fractal noise (`true`) or turbulence (`false`). The setting slightly alters the noise algorithm to produce a different style of noise with the same properties. Fractal noise is more continuous, whereas turbulence has more gaps.

- channelOptions — Which channels noise should be generated into. See the same option of noise(). Optional; defaults to RGB.

- grayscale — Whether or not the noise is grayscale. Optional; defaults to false.

- offsets — An array of Points, one for each octave, that position the layers.

The perlinNoise() method has a ton of options, but a couple of examples illustrate its capabilities.

Example 36-15 shows some basic Perlin noise and examines the effect of baseX, baseY, and numOctaves.

EXAMPLE 36-15 http://actionscriptbible.com/ch36/ex15

Basic Perlin Noise

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.events.KeyboardEvent;
  public class ch36ex15 extends Sprite {
    protected var bmp:BitmapData;
    protected var octaves:int = 1;
    public function ch36ex15() {
      bmp = new BitmapData(stage.stageWidth, stage.stageHeight);
      var bitmap:Bitmap = new Bitmap(bmp);
      stage.addChildAt(bitmap, 0);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    }
    protected function onKeyDown(event:KeyboardEvent):void {
      switch (String.fromCharCode(event.charCode)) {
        case "+": case "=": octaves++; break;
        case "-": case "_": octaves--; break;
      }
    }
    protected function onEnterFrame(event:Event):void {
      bmp.perlinNoise(mouseX, mouseY, octaves, 1, false, false);
      graphics.clear();
      graphics.lineStyle(0, 0xffffff, 0.5);
      graphics.drawRect(0, 0, mouseX, mouseY);
    }
  }
}
```

Try running the example. Move the mouse, and you'll update the size of the rectangle used to calculate a single cycle of the noise's first octave. Use the + and − keys to add and remove octaves. Notice that after around 5 octaves, you can't really see additional octaves; also note the impact on refresh rate.

In Example 36-16, you see the effect of edge stitching by creating a BitmapData object with Perlin noise and using it as a bitmap fill. Press a key on the keyboard to toggle stitching on and off.

765

Perlin Noise and Edge Stitching

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.events.KeyboardEvent;
  import flash.events.TimerEvent;
  [SWF(frameRate="2")]
  public class ch36ex16 extends Sprite {
    protected var bmp:BitmapData;
    protected var stitch:Boolean = false;
    public function ch36ex16() {
      bmp = new BitmapData(200, 200);
      var shape:Shape = new Shape();
      shape.graphics.lineStyle(0, 0, 0);
      shape.graphics.beginBitmapFill(bmp);
      shape.graphics.drawRect(0, 0, stage.stageWidth, stage.stageHeight);
      shape.graphics.endFill();
      addChild(shape);

      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onKeyDown(event:KeyboardEvent):void {
      stitch = !stitch;
    }
    private function onEnterFrame(event:Event):void {
      bmp.perlinNoise(100, 100, 2, Math.random()*1000, stitch, false, 1, true);
    }
  }
}
```

Using the offsets parameter to offset individual octaves of Perlin noise can generate interesting effects. For a good example, look ahead to Example 37-13 (http://actionscriptbible.com/ch37/ex13). In that example, you use Perlin noise to simulate wind rippling a flag. The bigger octaves move more slowly because the prevailing power of the wind fluctuates slowly, whereas smaller octaves move faster, pushing small ripples by much faster in addition to the overall motion.

Summary

- A bitmap is a grid of colors, one for each pixel. Bitmaps in Flash Player store 32 bits per pixel in four 8-bit channels: alpha, red, green, and blue.

- With bitmaps, you have total control over what goes on the screen. Use this to write anything from a simple effect to your own graphics engine.

- `BitmapData` stores the bitmap data. It doesn't give you direct access to the pixels in memory, but it allows you to retrieve and set them through certain methods.

- `BitmapData` can't be placed on the stage. `Bitmap` is a display object that can. `Bitmap` serves the sole purpose of displaying bitmap data.

- `BitmapData` objects can be transparent or discard alpha channel information. Set this with the `transparent` property.

- `BitmapData` objects are fixed in size at construction time.

- Bitmap data takes up memory. When releasing `BitmapData` instances, call `dispose()` on them to manually dispose the internal bitmap data.

- `Bitmaps` may be smoothly interpolated if the `Bitmap`'s `smoothing` property is set to `true`. If `false`, stage quality settings have no effect on their appearance.

- You can get the bitmap representation of any display object or composition of display objects by calling `draw()` on the `BitmapData` to draw into.

- Copy regions of bitmaps or channels of those regions with `copyPixels()` and `copyChannel()`.

- Access single pixels with the `[get/set]Pixel[32]()` family of methods.

- You can copy and set regions of — or whole — bitmaps in memory using a `ByteArray` or `Vector`.

- Fill a region with a solid color using `fillRect()`, or flood-fill areas of a solid color with `floodFill()`.

- You can apply filters and color transforms to a `BitmapData` with `applyFilter()` and `colorTransform()`; unlike with display objects, these have one-time effects.

- Methods especially useful for image analysis include `histogram()`, `threshold()`, and `getColorBoundsRect()`.

- Palette mapping — achieved with `paletteMap()` — is a powerful tool for color modification.

- Flash Player has built-in support for two kinds of procedural noise: fractal, naturalistic Perlin noise, and basic per-pixel noise.

Applying Filters

With ActionScript 3.0, you can easily apply one or a combination of preset graphical styles to any display object. Perhaps you'd like to blur an image, create a beveled edge, or add a glow to a `Sprite`, or maybe you want to distort an image. Like bitmap image-editing programs such as Photoshop, these treatments are made available as *filters*. In the Flash Player API, subclasses of the `flash.filters.BitmapFilter` class represent filters you can use to style display objects at runtime.

Introducing Filters

Filters are built-in effects you can apply to any `DisplayObject`, like `Sprite`, `MovieClip`, or `TextField`. All the filters discussed in this chapter are built into the Flash Player runtime, which makes them fast to render. Under the hood, when you apply a filter to a `DisplayObject`, Flash Player renders the display object as a bitmap and applies the filter to the bitmap to produce the desired graphical effect. Any time you apply a filter to a `DisplayObject`, its `cacheAsBitmap` property is set to `true`.

The filters available in Flash Player 9 and later are as follows:

- `BevelFilter`
- `BlurFilter`
- `ColorMatrixFilter`
- `ConvolutionFilter`
- `DisplacementMapFilter`
- `DropShadowFilter`
- `GlowFilter`
- `GradientBevelFilter`
- `GradientGlowFilter`

Flash Player 10 adds the `ShaderFilter`, enabling you to use Pixel Bender shaders easily on display objects with the same interface as any of the built-in filters. You'll learn about Pixel Bender shaders in Chapter 38, "Writing Shaders with Pixel Bender."

Applying Filters to Display Objects

Every `DisplayObject` instance has a `filters` property of type `Array`. This property contains the filter objects applied to the instance. If you apply a filter at authoring time in Flash Professional, that filter object is accessible during runtime via the `filters` property.

When you read elements from the `filters` array, you get copies of, rather than references to, the actual filter objects. That means that if you access an element in the `filters` array and change its properties, these changes to the filter aren't applied to the display object automatically. You've merely modified the properties of a new `BitmapFilter` instance that's not in use.

When you want to apply a filter to an object programmatically, you must assign an array of filter objects to the `filters` property. Not only does the `filters` array return copies of the filter objects, but when you read the `filters` property, it returns a copy of the array rather than a reference. So it won't work to use standard array methods such as `push()`, and you cannot overwrite elements of the array and have the updates affect the display object automatically. The only way to change, add, or remove filters is to assign a whole new array of filters to the `filters` property.

Some examples will help clarify this technique. In subsequent sections you'll learn more about each of the different types of filter classes. For the next few examples, however, you construct very basic filter objects. For example, the following code constructs a basic `DropShadowFilter` object:

```
var shadowFilter:DropShadowFilter = new DropShadowFilter();
```

You can only add filters to a `DisplayObject` by assigning a whole new `filters` array. So once the filter object is constructed, you pop it in a new array of filters. Here, I apply the drop shadow to a `DisplayObject` named `circle`:

```
circle.filters = [shadowFilter];
```

Additionally, if you change a property of a filter object after you assign it to the `filters` array of a display object, it will have no effect on the display object. For example, `DropShadowFilter` objects have a `distance` property that determines the displacement of the shadow in pixels. The default is 4 pixels. If you set the property before assigning the object to the `filters` array, there will be a visible effect:

```
shadowFilter.distance = 100;
circle.filters = [shadowFilter];
```

However, if you change the property after assigning the object to the filters array, it won't affect the display object:

```
circle.filters = [shadowFilter];
shadowFilter.distance = 100; //no effect
```

If you want to reapply a filter after changing the properties, as in the preceding example, you need to reassign the `filters` property. For example, the following retrieves a filter from the `filters` array, makes a change to it, and reassigns the whole array:

```
//get it
var shadowFilter:DropShadowFilter = circle.filters[0] as DropShadowFilter;
//change it
shadowFilter.distance = 0;
//reassign the whole thing
circle.filters = [shadowFilter];
```

Multiple Filters

Because the `filters` property is an array, you can add multiple filters on a single display object. The order of filters in the array determines the order that the graphic effects will be processed and can have a big effect on the final image. Of course, the more filters you use, the more processing will be required to render each frame. The size of images you're applying filters to and the quality setting of the filter both contribute to the speed at which Flash Player can render filtered display objects.

The next section looks more closely at each of the different filters and examines the parameters that each filter accepts and possible uses for each.

Blurs

You can add blur effects to display objects using `BlurFilter` objects. The `BlurFilter` class has three properties:

- `blurX` — The number of pixels to blur in the x direction. The default is 4.
- `blurY` — The number of pixels to blur in the y direction. The default is 4.
- `quality` — The number of times to run the blur. The range is from 0 to 15. The default value is 1.

You can make a new `BlurFilter` instance using the `BlurFilter` constructor. The constructor accepts from 0 to 3 parameters. The parameters are the same as in the preceding list. The following code constructs a `BlurFilter` instance with default settings:

```
var blurFilter:BlurFilter = new BlurFilter();
```

The following constructs a new `BlurFilter` instance that blurs 100 pixels in the x direction but no pixels in the y direction. It uses a quality setting of 15 to achieve a very smooth blur.

```
var blurFilter:BlurFilter = new BlurFilter(100, 0, 15);
```

As with any other filter, you can change the properties of a `BlurFilter` object after it has been instantiated.

Example 37-1 uses blur filters on a stack of ball-like objects layered in Z space to simulate a camera lens with a narrow depth of field. This example requires Flash Player 10 and later but can be easily retrofitted for Flash Player 9 by substituting a scaleX and scaleY factor for the z property.

EXAMPLE 37-1 http://actionscriptbible.com/ch37/ex1

Using the Blur Filter

```
package {
  import flash.display.GradientType;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.filters.BlurFilter;
  import flash.geom.Matrix;

  public class ch37ex1 extends Sprite {
    protected const NUM_COPIES:int = 28;
    public function ch37ex1() {
      var W:Number = stage.stageWidth;
      var H:Number = stage.stageHeight;
      for (var i:int = 0; i < NUM_COPIES; i++) {
        var s:Sprite = new Sprite();
        var m:Matrix = new Matrix();
        m.createGradientBox(40, 40, 0, -20, -20);
        s.graphics.beginGradientFill(GradientType.RADIAL, [0xE76921, 0x8F1F08],
          [1, 1], [0, 255], m, null, null, 0.6);
        s.graphics.drawCircle(0, 0, 20);
        s.graphics.endFill();
        s.x = (Math.random()-0.5)*200 + 0.5*W;
        s.y = (Math.random()-0.5)*200 + 0.5*H;
        s.z = -(i-NUM_COPIES/2) * W/NUM_COPIES;
        s.filters = [new BlurFilter(0, 0, 2)];
        addChild(s);
      }
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      var focusZ:Number = stage.mouseX - stage.stageWidth/2;
      for (var i:int = 0; i < numChildren; i++) {
        var s:Sprite = Sprite(getChildAt(i));
        var blurFilter:BlurFilter = BlurFilter(s.filters[0]);
        var blurAmount:Number = 0.05 * Math.abs(focusZ - s.z);
        blurFilter.blurX = blurAmount;
        blurFilter.blurY = blurAmount;
        s.filters = [blurFilter];
      }
    }
  }
}
```

Drop Shadows

You can use a `DropShadowFilter` to apply a drop shadow to a display object. Instances of `DropShadowFilter` have the following properties:

- `distance` — The number of pixels from the display object to offset the shadow. The default is 4.

- `angle` — The angle of the light source in degrees, where 0 is directly to the left of the object. The default value is 45.

- `color` — The color of the shadow specified as an unsigned integer from 0x000000 to 0xFFFFFF. The default value is 0x000000.

- `alpha` — The alpha of the shadow from 0 to 1. The default is 1.

- `blurX` — The amount to blur the shadow in the x direction from 0 to 255. The default is 4.

- `blurY` — The amount to blur the shadow in the y direction from 0 to 255. The default is 4.

- `strength` — The punch strength of the shadow from 0 to 255. The default is 1.

- `quality` — The number of times to run the blur on the shadow. The range is from 0 to 15. The higher the number, the smoother the blur of the shadow will appear. The default is 1.

- `inner` — A Boolean value indicating whether the shadow should be applied to the inside of the object. The default value is `false`, which means the shadow is applied outside the object.

- `knockout` — A Boolean value indicating whether the original display object contents should be transparent. If `true`, that section of the object is transparent (the background or whatever is underneath the object is visible), and the drop shadow is visible around the edges of that area. The default value is `false`.

- `hideObject` — A Boolean value indicating whether to hide the contents of the display object. If `true`, the contents of the display object are hidden, and the drop shadow is visible. The default value is `false`.

The `DropShadowFilter` constructor accepts between 0 and 11 parameters. The parameters are in the order they appear in the preceding list. The following constructs a `DropShadowFilter` object with the default settings:

```
var shadowFilter:DropShadowFilter = new DropShadowFilter();
```

The following constructs a `DropShadowFilter` object with a red shadow offset of 20 pixels to the right:

```
var shadowFilter:DropShadowFilter = new DropShadowFilter(20, 0, 0xFF0000);
```

Example 37-2 simulates a light source with the mouse cursor, illuminating some text in the center and projecting the shadow away from the light source.

`http://actionscriptbible.com/ch37/ex2`

Drop Shadows

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.filters.DropShadowFilter;
  import flash.geom.Point;
  import flash.text.*;

  public class ch37ex2 extends Sprite {
    protected var s:Sprite;
    public function ch37ex2() {
      var tf:TextField = new TextField();
      tf.defaultTextFormat = new TextFormat("_sans", 35, 0, true, false);
      tf.selectable = false;
      tf.width = tf.height = 0;
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.text = "Hello Shadows";
      s = new Sprite();
      s.addChild(tf);
      tf.x = -tf.textWidth/2;
      tf.y = -tf.textHeight/2;
      s.x = stage.stageWidth/2;
      s.y = stage.stageHeight/2;
      addChild(s);

      s.filters = [new DropShadowFilter(0, 0, 0, 0.4, 12, 12, 1, 2)];
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      var filter:DropShadowFilter = DropShadowFilter(s.filters[0]);
      var vector:Point = new Point(stage.mouseX - s.x, stage.mouseY - s.y);
      var m:Number = new Point(stage.stageWidth/2, stage.stageHeight/2).length;
      var normalizedDistance:Number = vector.length / m;
      filter.distance = Math.pow(normalizedDistance, 2) * 100;
      filter.blurX = filter.blurY = normalizedDistance * 20;
      filter.alpha = Math.pow(normalizedDistance, 2);
      filter.angle = Math.atan2(vector.y, vector.x) / Math.PI * 180 +  180;
      s.filters = [filter];
    }
  }
}
```

Bevels

A bevel filter simulates a beveled edge on top of the content it's filtering. It makes the surface appear as though the edge has been chiseled to an angle, so it catches the light. Bevel filters are represented by instances of the `BevelFilter` class, which has the following properties:

- `distance` — The offset of the bevel in pixels. The greater the number, the more pronounced the bevel. The default is 4.

- `angle` — The angle of the light source to the object in degrees. A value of 0 makes the light source directly to the left, whereas a value of 180 makes the light source directly to the right. The default is 45.

- `highlightColor` — The highlight color specified as an unsigned integer from 0x000000 to 0xFFFFFF. The default value is white (0xFFFFFF).

- `highlightAlpha` — The alpha of the highlight. The range is from 0 to 1. The default is 1.

- `shadowColor` — The shadow color specified as an unsigned integer from 0x000000 to 0xFFFFFF. The default value is black (0x000000).

- `shadowAlpha` — The alpha of the shadow. The range is from 0 to 1. The default is 1.

- `blurX` — The number of pixels to blur the bevel in the horizontal direction. The greater the number, the softer the bevel appears. The default is 4.

- `blurY` — The number of pixels to blur the bevel in the vertical direction. The greater the number, the softer the bevel appears. The default is 4.

- `strength` — The punch strength of the bevel. The greater the number, the more visible the shadow and highlight will be when the blur properties are set to greater numbers. The default is 1.

- `quality` — The number of times to run the blur. The range is from 1 to 15. The greater the number, the smoother the bevel appears. The default is 1.

- `type` — The type can be one of the constants `BitmapFilterType.INNER`, `BitmapFilterType.OUTER`, or `BitmapFilterType.FULL`. The default is `BitmapFilterType.INNER`.

- `knockout` — Whether to knock out the contents of the object. The default is `false`.

The `BevelFilter` constructor accepts between 0 and 12 parameters. Parameters are in the same order they appear in the preceding list. For example, the following instantiates a `BevelFilter` object with a distance of 10 pixels and an angle of 0:

```
var bevelFilter:BevelFilter = new BevelFilter(10, 0);
```

The following instantiates a new `BevelFilter` object with each of the 12 parameters:

```
var bevelFilter:BevelFilter = new BevelFilter(10, 0,
  0xFF0000, .5, 0xCCCCCC, .5, 10, 10, 5, 15,
  BitmapFilterType.INNER, true);
```

Example 37-3 adds onto the effects in the previous example with a bevel also following the virtual light under the mouse.

775

EXAMPLE 37-3 `http://actionscriptbible.com/ch37/ex3`

Bevel Filters

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.filters.BevelFilter;
  import flash.filters.DropShadowFilter;
  import flash.geom.Point;
  import flash.text.*;

  public class ch37ex3 extends Sprite {
    protected var s:Sprite;
    public function ch37ex3() {
      var tf:TextField = new TextField();
      tf.defaultTextFormat = new TextFormat("_sans", 35, 0x303030, true, false);
      tf.selectable = false;
      tf.width = tf.height = 0;
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.text = "Hello Bevels";
      s = new Sprite();
      s.addChild(tf);
      tf.x = -tf.textWidth/2;
      tf.y = -tf.textHeight/2;
      s.x = stage.stageWidth/2;
      s.y = stage.stageHeight/2;
      addChild(s);

      s.filters = [new DropShadowFilter(0, 0, 0, 0.4, 12, 12, 1, 2),
                   new BevelFilter(2, 0)];
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      var shadow:DropShadowFilter = DropShadowFilter(s.filters[0]);
      var bevel:BevelFilter = BevelFilter(s.filters[1]);
      var vector:Point = new Point(stage.mouseX - s.x, stage.mouseY - s.y);
      var m:Number = new Point(stage.stageWidth/2, stage.stageHeight/2).length;
      var normalizedDistance:Number = vector.length / m;
      shadow.distance = Math.pow(normalizedDistance, 2) * 30;
      shadow.blurX = shadow.blurY = normalizedDistance * 20;
      shadow.alpha = Math.pow(normalizedDistance, 2);
      shadow.angle = Math.atan2(vector.y, vector.x) / Math.PI * 180 +  180;
      bevel.angle = shadow.angle;
      bevel.strength = 2 * normalizedDistance;
      s.filters = [shadow, bevel];
    }
  }
}
```

Gradient Bevels

The `GradientBevelFilter` class is much like the `BevelFilter` class, except that rather than adding areas of highlight and areas of shadow to the display object, the entire area is filled with gradations between the values you provide. The class defines the following properties:

- `distance` — The number of pixels from the edge of the display object to offset the bevel. The default is 4.

- `angle` — The angle of the light source in degrees, where 0 is directly to the left of the object. The default value is 45.

- `colors` — The colors of the bevel specified as an array of an unsigned integer from 0x000000 to 0xFFFFFF.

- `alphas` — The alphas of the bevel specified as an array of values from 0.0 to 1.0 such that each element corresponds to an element in the colors array. The default is `null`.

- `ratios` — The ratios to use when distributing the colors across the bevel specified as an array of values from 0 to 255. Each element of the array corresponds to an element of the colors array. Each color in the colors array blends into the next, so each color appears at 100 percent at just one point in the continuum.

- `blurX` — The amount to blur the bevel in the x direction from 0 to 255. The default is 4.

- `blurY` — The amount to blur the bevel in the y direction from 0 to 255. The default is 4.

- `strength` — The punch strength of the bevel from 0 to 255. The default is 1.

- `quality` — The number of times to run the blur on the bevel. The range is from 0 to 15. The higher the number, the smoother the blur of the bevel appears. The default is 1.

- `type` — The type can be one of the constants `BitmapFilterType.INNER`, `BitmapFilterType.OUTER`, or `BitmapFilterType.FULL`. The default is `BitmapFilterType.INNER`.

- `knockout` — A Boolean value indicating whether the original display object contents should be transparent. If `true`, that section of the object is transparent. (The background or whatever is underneath the object is visible.) The default value is `false`.

The `GradientBevelFilter` constructor accepts between 0 to 11 parameters from the preceding list. The following code constructs a new `GradientBevelFilter` with the default values:

```
var gradientBevel:GradientBevelFilter = new GradientBevelFilter();
```

Unlike many other filter types, the `GradientBevelFilter` default values don't have a visible effect because the defaults for the three array parameters are undefined. Therefore, for there to be some visible effect, you need to specify values for at least the three array parameters. The following code constructs a new `GradientBevelFilter` object with blue, white, and black colors:

```
var colors:Array = [0x0000FF, 0xFFFFFF, 0xFFFFFF,
                    0xFFFFFF, 0x000000];
var alphas:Array = [100, 20, 0, 20, 100];
var ratios:Array = [0, 255 / 4, 2 * 255 / 4, 3 * 255 / 4, 255];
var gradientBevel:GradientBevelFilter = new GradientBevelFilter(10, 45,
  colors, alphas, ratios);
```

Bevels, including gradient bevels, are great for making faux-3D objects. Example 37-4 exploits `GradientBevelFilter` to produce a decent-looking button using only code.

EXAMPLE 37-4   http://actionscriptbible.com/ch37/ex4

Creating 3D-Looking Objects with Bevels

```
package {
  import flash.display.Sprite;
  public class ch37ex4 extends Sprite {
    public function ch37ex4() {
      var b:BeveledButton;
      b = new BeveledButton("Click me");
      b.x = b.y = 100;
      addChild(b);
      b = new BeveledButton("Please don't click me");
      b.x = 200; b.y = 100;
      addChild(b);
    }
  }
}
import flash.display.*;
import flash.text.*;
import flash.filters.GradientBevelFilter;
import flash.events.MouseEvent;
class BeveledButton extends Sprite {
  protected var _label:String;
  protected var tf:TextField;
  protected var bg:Shape;
  protected const ANGLE:Number = -10;
  protected const MARGIN_W:Number = 15, MARGIN_H:Number = 8;
  public function BeveledButton(initialLabel:String = "") {
    bg = new Shape();
    addChild(bg);
    tf = new TextField();
    tf.defaultTextFormat = new TextFormat("_sans", 11, 0xffffff, true);
    tf.selectable = false;
    tf.multiline = false;
    tf.autoSize = TextFieldAutoSize.LEFT;
    addChild(tf);

    mouseChildren = false;
    buttonMode = true;

    var colors:Array = [0x606060, 0x909090, 0xe0e0e0];
    var alphas:Array = [100, 100, 100];
    var ratios:Array = [0, 127, 255];
    bg.filters=[new GradientBevelFilter(3, 90+ANGLE, colors, alphas, ratios)];
```

```
    label = initialLabel;
    addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
  }
  public function set label(s:String):void {
    tf.width = tf.height = 0;
    tf.text = s;
    var W:Number = tf.textWidth + 2*MARGIN_W;
    var H:Number = tf.textHeight - 4 + 2*MARGIN_H;
    tf.y = MARGIN_H - 4;
    tf.x = MARGIN_W;
    bg.graphics.clear();
    bg.graphics.beginFill(0);
    bg.graphics.drawRoundRect(0, 0, W, H, 8);
    bg.graphics.endFill();
  }
  public function get label():String {return _label;}
  protected function onMouseDown(event:MouseEvent):void {
    var bevel:GradientBevelFilter = GradientBevelFilter(bg.filters[0]);
    bevel.angle = -90 + ANGLE;
    bg.filters = [bevel];
    tf.x += 1;
    tf.y += 1;
  }
  protected function onMouseUp(event:MouseEvent):void {
    var bevel:GradientBevelFilter = GradientBevelFilter(bg.filters[0]);
    bevel.angle = 90 + ANGLE;
    bg.filters = [bevel];
    tf.x -= 1;
    tf.y -= 1;
  }
}
```

Try modifying this code to include more colors in the gradient. See how that affects the appearance of the buttons.

Glows

You can use instances of the GlowFilter class to make display objects glow either from the inside or to the outside. The GlowFilter class has the following properties:

- color — The color of the glow specified as an unsigned integer from 0x000000 to 0xFFFFFF. The default value is 0xFF0000.
- alpha — The alpha of the glow from 0 to 1. The default is 1.

- `blurX` — The amount to blur the glow in the x direction from 0 to 255. The default is 6.
- `blurY` — The amount to blur the glow in the y direction from 0 to 255. The default is 6.
- `strength` — The punch strength of the glow from 0 to 255. The default is 2.
- `quality` — The number of times to run the blur on the glow. The range is from 0 to 15. The higher the number, the smoother the blur of the glow appears. The default is 1.
- `inner` — A Boolean value indicating whether the glow should be applied to the inside of the object. The default value is `false`, which means the glow is applied outside the object.
- `knockout` — A Boolean value indicating whether the original display object contents should be transparent. If `true`, that section of the object is transparent (the background or whatever is underneath the object is visible), and the glow is visible around the edges of that area. The default value is `false`.

The `GlowFilter` constructor accepts from 0 to 8 parameters as described in the list. The parameters are in the order they appear in the preceding list.

I might have gone a bit overboard with Example 37-5. A glow filter is perfect to simulate a flashing light. So this example re-creates the classic game "Simon," in which there are four color buttons that light up in a sequence, which you must match. You and the computer take turns, the sequence getting longer by one move every time you successfully repeat it, as well as playing faster. The flashing lights are simulated by an outer glow with an animating `alpha` value. Each colored light has a similarly colored glow. Additionally, the example incorporates audio synthesis, each button having a unique tone. See Chapter 31, "Playing and Generating Sound," for more.

EXAMPLE 37-5   `http://actionscriptbible.com/ch37/ex5`

A Game Using Glow Filters

```
package {
  import flash.display.Sprite;
  [SWF(backgroundColor="#000000")]
  public class ch37ex5 extends Sprite {
    public function ch37ex5() {
      var game:RepeatGame = new RepeatGame(stage);
      addChild(game);
      game.x = stage.stageWidth/2;
      game.y = stage.stageHeight/2;
      game.startGame();
    }
  }
}
import flash.filters.GlowFilter;
import flash.display.*;
import flash.events.*;
import flash.utils.*;
import flash.ui.Keyboard;
import flash.media.Sound;
class RepeatGame extends Sprite {
  protected var cpuMoves:Array;
```

```
  protected var moveIndex:int;
  protected var isCpuTurn:Boolean;
  protected var movePlaybackTimer:Timer;
  protected var piecesByKeyCode:Array;
  protected var piecesById:Array;
  public function RepeatGame(s:Stage) {
    s.addEventListener(KeyboardEvent.KEY_DOWN, onPlayerMove);
    movePlaybackTimer = new Timer(0);
    movePlaybackTimer.addEventListener(TimerEvent.TIMER, onCpuMove);
    buildPieces();
  }
  protected function buildPieces():void {
    piecesById = new Array();
    piecesByKeyCode = new Array();
    var piece:GamePiece;
    var arci:Number = - 3/4 * Math.PI;
    piece = new GamePiece(0x47cf51, 0x00ff00, arci, arci + Math.PI/2, 220);
    piecesByKeyCode[Keyboard.UP] = piece;
    piecesById.push(piece);
    addChild(piece);
    arci += Math.PI/2;
    piece = new GamePiece(0xe52a37, 0xff0000, arci, arci + Math.PI/2, 293);
    piecesByKeyCode[Keyboard.RIGHT] = piece;
    piecesById.push(piece);
    addChild(piece);
    arci += Math.PI/2;
    piece = new GamePiece(0x3e49da, 0x0000ff, arci, arci + Math.PI/2, 392);
    piecesByKeyCode[Keyboard.DOWN] = piece;
    piecesById.push(piece);
    addChild(piece);
    arci += Math.PI/2;
    piece = new GamePiece(0xf5ed10, 0xffff00, arci, arci + Math.PI/2, 440);
    piecesById.push(piece);
    piecesByKeyCode[Keyboard.LEFT] = piece;
    addChild(piece);
  }
  public function startGame():void {
    movePlaybackTimer.delay = 2000;
    cpuMoves = new Array();
    nextRound();
  }
  protected function nextRound():void {
    trace("NEXT ROUND!");
    isCpuTurn = true
    cpuMoves.push(int(Math.random() * piecesById.length));
    moveIndex = 0;
    movePlaybackTimer.delay *= 0.75;
    movePlaybackTimer.reset();
    movePlaybackTimer.start();
  }
```

continued

EXAMPLE 37-5 *(continued)*

```
  protected function onCpuMove(event:TimerEvent):void {
    GamePiece(piecesById[cpuMoves[moveIndex]]).flash();
    if (++moveIndex >= cpuMoves.length) {
      isCpuTurn = false;
      moveIndex = 0;
      movePlaybackTimer.stop();
      return;
    }
  }
  protected function onPlayerMove(event:KeyboardEvent):void {
    var selectedPiece:GamePiece = piecesByKeyCode[event.keyCode] as GamePiece;
    if (selectedPiece == null || isCpuTurn) return;
    selectedPiece.flash();
    var correct:Boolean = (selectedPiece == piecesById[cpuMoves[moveIndex]]);
    if (correct) {
      trace("RIGHT!");
      if (++moveIndex >= cpuMoves.length) {
        isCpuTurn = true;
        setTimeout(nextRound, movePlaybackTimer.delay);
      }
    } else {
      trace("WRONG!");
      isCpuTurn = true;
      setTimeout(startGame, 1000);
    }
  }
}
class GamePiece extends Sprite {
  protected const INNER_RADIUS:Number = 100;
  protected const OUTER_RADIUS:Number = 160;
  protected const ARC_PADDING:Number = 0.06;
  protected var glow:GlowFilter;
  protected var pitchHz:Number;
  public function GamePiece(color:uint, glowColor:uint,
                            arcStartAngle:Number, arcEndAngle:Number,
                            pitchHz:Number) {
    this.pitchHz = pitchHz;
    arcStartAngle += ARC_PADDING;
    arcEndAngle -= ARC_PADDING;
    graphics.beginFill(color);
    var step:Number = (arcEndAngle - arcStartAngle) / 200;
    var theta:Number;
    for (theta = arcStartAngle; theta <= arcEndAngle; theta += step) {
      var X:Number = Math.cos(theta) * INNER_RADIUS;
      var Y:Number = Math.sin(theta) * INNER_RADIUS;
      if (theta == arcStartAngle) {
        graphics.moveTo(X, Y);
      } else {
        graphics.lineTo(X, Y);
      }
```

```
    }
    for (theta = arcEndAngle; theta >= arcStartAngle; theta -= step) {
      graphics.lineTo(Math.cos(theta) * OUTER_RADIUS,
        Math.sin(theta) * OUTER_RADIUS);
    }
    graphics.endFill();
    glow = new GlowFilter(glowColor, 0, 32, 32, 2, 2, false);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  public function flash():void {
    glow.alpha = 1;
    var s:Sound = new Sound();
    s.addEventListener(SampleDataEvent.SAMPLE_DATA, onBufferSound);
    s.play();
  }
  protected function onBufferSound(event:SampleDataEvent):void {
    var tEnd:int = 44100 * 0.5; //duration = 0.5sec
    var tMax:int = Math.min(tEnd, event.position + 8192);
    for (var t:int = event.position; t < tMax; t++) {
      var n:Number = Math.sin(t / 44100 * 2 * Math.PI * pitchHz);
      n *= ((tEnd-t) / tEnd) * 0.5; //fade out
      event.data.writeFloat(n);
      event.data.writeFloat(n);
    }
  }
  protected function onEnterFrame(event:Event):void {
    if (glow.alpha > 0) {
      glow.alpha -= 0.05;
      filters = [glow];
    } else {
      filters = null;
    }
  }
}
```

Play this game with the arrow keys on your keyboard. Try forking the code to include more possible buttons or to add more effects to the buttons.

Gradient Glows

Gradient glows are applied using the GradientGlowFilter class. This renders the glow as a custom gradient instead of the default single-color alpha ramp of a normal glow. The properties of the GradientGlowFilter class are identical to those of GradientBevelFilter. Similarly, the GradientGlowFilter constructor accepts the parameters in the same order that the GradientBevelFilter constructor does.

Color Effects

You can achieve some simple color effects without the use of filters by altering the `ColorTransform` of a display object, as described in Chapter 34, "Geometric and Color Transformations." One adaptable filter, however, gives you much richer control over the colors in a display object. Using a `ColorMatrixFilter`, you can adjust the saturation, hue, or contrast of a display object or apply other special color effects. The `ColorMatrixFilter` changes every pixel's color by performing matrix multiplication on the pixel's color and a matrix of your design.

When using a color matrix, the color of each pixel is decomposed into its red, green, blue, and alpha components and packed into a 5×1 column matrix:

```
| red   |
| green |
| blue  |
| alpha |
| 1     |
```

The matrix you provide a `ColorMatrixFilter` is 4×5. This is multiplied by the input color, resulting in the output color:

```
| a b c d e |   | R |   | output red   |
| f g h i j |   | G |   | output green |
| k l m n o |   | B | = | output blue  |
| p q r s t |   | A |   | output alpha |
                | 1 |
```

Multiplying the 4×5 filter matrix by the 5×1 input color matrix results in a 4×1 output color matrix. If you do the matrix multiplication by hand, you see that each channel of the output color is a linear combination of the input color channels. In other words, by choosing the contents of the filter matrix you create equations that define each output color channel as a combination of the input colors. Now you'll expand the matrix multiplication to see that in action:

```
output red   = aR + bG + cB + dA + e
output green = fR + gG + hB + iA + j
output blue  = kR + lG + mB + nA + o
output alpha = pR + qG + rB + sA + t
```

The power of this method of transforming colors comes from the fact that you can tweak just how each channel relies on the other ones. The extra 1 that pads the input color matrix allows for the addition of a fifth column of the filter matrix that can add a color offset to each channel that is independent of the input color. You can see that the column `[e, j, o, t]` and the terms e, j, o, and t in the preceding equations are not multiplied by any of the input color channels.

When using a `ColorMatrixFilter`, you specify the 4×5 matrix in the `matrix` property of the `ColorMatrixFilter` instance. Instead of using a two-dimensional array, you should provide the matrix contents as a one-dimensional array in row-major order. That is, lay out the matrix terms left to right and top to bottom in an `Array` of size 20:

```
var cmt:ColorMatrixTransform = new ColorMatrixTransform();
cmt.matrix = [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t];
```

Although you will provide the matrix's contents in a flat array, remember that they represent and will be interpreted as a 4×5 matrix.

The default filter matrix is almost an identity matrix; when it is multiplied by the input color matrix, the colors remain unchanged. (The filter matrix can't be a real identity matrix because it's nonsquare.)

```
| 1 0 0 0 0 |  | R |    | R |
| 0 1 0 0 0 |  | G |    | G |
| 0 0 1 0 0 |  | B | =  | B |
| 0 0 0 1 0 |  | A |    | A |
              | 0 |
```

By designing the filter matrix carefully, you can achieve many interesting effects. Following are some of the more useful color effects. I've tried to organize these so that the matrix multiplications start out simple — even if you don't follow along with the matrix math, you should be able to get a feel for how the parameters in the filter matrix affect the colors in the image.

Note

The `Adjust Color` filter available in Flash Professional uses a `ColorMatrixFilter` to achieve the brightness, contrast, saturation, and hue adjustments shown in the filter's UI. ■

Brightness

You can change the brightness of an image by either scaling or offsetting the color channels uniformly. Example 37-6 compares an unfiltered original image, a version in which each of the red, green, and blue values is uniformly doubled, and a version in which each of the red, green, and blue values is uniformly shifted up by 60. These appear left to right in the example.

EXAMPLE 37-6 `http://actionscriptbible.com/ch37/ex6`

Adjusting Brightness with a ColorMatrixFilter

```
package {
  import flash.display.Sprite;
  import flash.filters.ColorMatrixFilter;

  public class ch37ex6 extends Sprite {
    function ch37ex6() {
      var original:TestImage = new TestImage(0.3);
      var a:TestImage = new TestImage(0.3, 150);
      var b:TestImage = new TestImage(0.3, 300);
      addChild(original);
      addChild(a);
      addChild(b);
      //brighten method 1: scale
```

continued

EXAMPLE 37-6 *(continued)*

```
      var aMatrix:Array = [2, 0, 0, 0, 0,
                           0, 2, 0, 0, 0,
                           0, 0, 2, 0, 0,
                           0, 0, 0, 1, 0];
      a.filters = [new ColorMatrixFilter(aMatrix)];
      //brighten method 2: offset
      var bMatrix:Array = [1, 0, 0, 0, 60,
                           0, 1, 0, 0, 60,
                           0, 0, 1, 0, 60,
                           0, 0, 0, 1,  0];
      b.filters = [new ColorMatrixFilter(bMatrix)];
    }
  }
}
import flash.display.Loader;
import flash.net.URLRequest;
import flash.system.LoaderContext;
class TestImage extends Loader {
  public function TestImage(scale:Number = 1, x:Number = 0, y:Number = 0) {
    //photo    (CC-BY) Roger Braunstein
    //source   http://www.flickr.com/photos/rogerimp/2940373537/
    var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
    load(new URLRequest(url), new LoaderContext(true));
    scaleX = scaleY = scale;
    this.x = x; this.y = y;
  }
}
```

Tint

You can use a color matrix to tint an image. There are two ways to apply the tint, and they correspond to the two ways of tinting via a `ColorTransform` object: multipliers and offsets. Using a `ColorMatrixFilter` object, the multipliers are the diagonal elements represented by r, g, b, and a in the following matrix.

```
| r 0 0 0 0 |
| 0 g 0 0 0 |
| 0 0 b 0 0 |
| 0 0 0 a 0 |
```

The range that is typically useful for the multipliers is from −1 to 1. Changing the multipliers in a matrix that otherwise looks like an identity matrix is called *scaling* the color because the effect of such a matrix is that it simply multiplies one or more of the color components (red, blue, green, or alpha).

You can also apply offsets to each of the color components via a matrix in the following form.

```
| 1 0 0 0 r |
| 0 1 0 0 g |
| 0 0 1 0 b |
| 0 0 0 1 a |
```

In this matrix, r, g, b, and a represent the red, green, blue, and alpha offsets. The range that is typically useful for offsets is from −255 to 255.

Example 37-7 compares an unmodified original image, one tinted by scaling down the green and blue channels to emphasize the red channel, and one tinted by topping out the red channel with an offset.

EXAMPLE 37-7 http://actionscriptbible.com/ch37/ex7

Tinting with a ColorMatrixFilter

```
package {
  import flash.display.Sprite;
  import flash.filters.ColorMatrixFilter;

  public class ch37ex7 extends Sprite {
    function ch37ex7() {
      var original:TestImage = new TestImage(0.3);
      var a:TestImage = new TestImage(0.3, 150);
      var b:TestImage = new TestImage(0.3, 300);
      addChild(original);
      addChild(a);
      addChild(b);
      //tint method 1: scale down G & B
      var aMatrix:Array = [1,   0,   0, 0, 0,
                           0, 0.5,   0, 0, 0,
                           0,   0, 0.5, 0, 0,
                           0,   0,   0, 1, 0];
      a.filters = [new ColorMatrixFilter(aMatrix)];
      //tint method 2: shift up R
      var bMatrix:Array = [1, 0, 0, 0, 255,
                           0, 1, 0, 0,   0,
                           0, 0, 1, 0,   0,
                           0, 0, 0, 1,   0];
      b.filters = [new ColorMatrixFilter(bMatrix)];
    }
  }
}
import flash.display.Loader;
import flash.net.URLRequest;
import flash.system.LoaderContext;
class TestImage extends Loader {
  public function TestImage(scale:Number = 1, x:Number = 0, y:Number = 0) {
    //photo   (CC-BY) Roger Braunstein
    //source  http://www.flickr.com/photos/rogerimp/2940373537/
    var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
```

continued

EXAMPLE 37-7 *(continued)*

```
    load(new URLRequest(url), new LoaderContext(true));
    scaleX = scaleY = scale;
    this.x = x; this.y = y;
  }
}
```

Of course, you can combine both multipliers and offsets in one matrix, just as you can combine multipliers and offsets in a `ColorTransform` object.

Negative

A digital negative substitutes the complementary color for each pixel in the original. A color and its complement always add up to white (`0xFFFFFF`). It follows that each channel of a color and the corresponding channel from its inverse add up to `0xFF` or 255. You can calculate the complementary color by subtracting the red, blue, and green parts each from 255. See if you can figure out the matrix needed to find every color's complement. For the moment just worry about the red channel. Call the input red component `R` and the output red component `R'`.

```
R' + R = 255
R' = 255 - R
R' = -R + 0G + 0B + 255
```

Extrapolate or repeat for the green and blue channels, and you'll realize the matrix must be

```
| -1  0  0 0 255 |
|  0 -1  0 0 255 |
|  0  0 -1 0 255 |
|  0  0  0 1   0 |
```

Example 37-8 compares a test image and its digital negative.

EXAMPLE 37-8 http://actionscriptbible.com/ch37/ex8

Applying a Negative with a ColorMatrixFilter

```
package {
  import flash.display.Sprite;
  import flash.filters.ColorMatrixFilter;

  public class ch37ex8 extends Sprite {
    function ch37ex8() {
      var original:TestImage = new TestImage(0.5);
      var neg:TestImage = new TestImage(0.5, 250);
      addChild(original);
```

```
        addChild(neg);
        var negMatrix:Array = [-1,  0,  0, 0, 255,
                                0, -1,  0, 0, 255,
                                0,  0, -1, 0, 255,
                                0,  0,  0, 1,   0];
        neg.filters = [new ColorMatrixFilter(negMatrix)];
      }
    }
  }
  import flash.display.Loader;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  class TestImage extends Loader {
    public function TestImage(scale:Number = 1, x:Number = 0, y:Number = 0) {
      //photo   (CC-BY) Roger Braunstein
      //source  http://www.flickr.com/photos/rogerimp/2940373537/
      var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
      load(new URLRequest(url), new LoaderContext(true));
      scaleX = scaleY = scale;
      this.x = x; this.y = y;
    }
  }
```

Contrast

You can adjust the contrast of a display object by applying a `ColorMatrixFilter` object that both scales *and* offsets the color channels. To increase contrast, scale up all the colors but then compensate for the additional brightness with a negative offset. Likewise, to decrease the contrast, scale down the colors but then shift them up to keep them from getting too dark.

In Example 37-9, you can modify the contrast by sliding the mouse from left to right.

EXAMPLE 37-9 `http://actionscriptbible.com/ch37/ex9`

Changing Contrast with a ColorMatrixFilter

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.filters.ColorMatrixFilter;

  public class ch37ex9 extends Sprite {
    protected var img:TestImage;
    function ch37ex9() {
      img = new TestImage();
      addChild(img);
```

EXAMPLE 37-9 *(continued)*

```
    addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
  }
  protected function onMouseMove(event:MouseEvent):void {
    //1 for no change, 0 for lowest contrast, 2 and up for high contrast
    var contrastAmount:Number;
    //mouse at left for lowest contrast, right for 3x contrast boost
    contrastAmount = stage.mouseX / stage.stageWidth * 3;
    var scale:Number = contrastAmount;
    var offset:Number = 128 * (1 - scale);
    var aContrast:Array = [scale,     0,     0, 0, offset,
                               0, scale,     0, 0, offset,
                               0,     0, scale, 0, offset,
                               0,     0,     0, 1,      0];
    img.filters = [new ColorMatrixFilter(aContrast)];
  }
  }
}
import flash.display.Loader;
import flash.net.URLRequest;
import flash.system.LoaderContext;
class TestImage extends Loader {
  public function TestImage(scale:Number = 1, x:Number = 0, y:Number = 0) {
    //photo   (CC-BY) Roger Braunstein
    //source  http://www.flickr.com/photos/rogerimp/2940373537/
    var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
    load(new URLRequest(url), new LoaderContext(true));
    scaleX = scaleY = scale;
    this.x = x; this.y = y;
  }
}
```

Convert to Grayscale

You can design a matrix that will remove all color from an image. Try deriving it from first principles.

Shades of gray contain equal amounts of red, green, and blue: black has 0 of each, 50 percent gray has 128 of each (0x808080, 0x80 = 128), and white has 255 of each (0xFFFFFF, 0xFF = 255). So one property of the matrix is that all the equations to determine red, green, and blue channels must be the same, to ensure the output is equal. In matrix terms, this means the first three rows should be identical.

In a grayscale image, there is no hue. There is only the relative lightness or darkness of the pixel. So if you wanted to convert a color image to grayscale, the color of the output pixel should be related to the brightness of the input pixel. A good way to do this is to average the RGB channels. So every gray value is determined by an equal weighting of the brightness of the three color channels. This leads to the simple equation:

```
R' = G' = B' = (R * G * B)/3 = 0.33R + 0.33G + 0.33G
```

Or, in matrix form:

```
| 0.33 0.33 0.33 0 0 |
| 0.33 0.33 0.33 0 0 |
| 0.33 0.33 0.33 0 0 |
|   0    0     0 1 0 |
```

However, brains are wired to perceive visual reality in a subjective manner. Anyone who's looked at optical illusions knows how subjective vision is. Sometimes what is mathematically precise is not what looks the best. Remember in Chapter 35, "Programming Vector Graphics," that the default blending mode for gradients is not linear through RGB — there, the blending mode that looks best isn't the most mathematically precise. Likewise, you can make a better-looking grayscale image by choosing nonequal bias for each of the red, green, and blue channels. Choosing biases, or *luminance coefficients*, of 0.3086, 0.6094, and 0.0820, respectively, yields a more natural-looking image. This matrix encodes the constants:

```
| 0.3086 0.6094 0.0820 0 0 |
| 0.3086 0.6094 0.0820 0 0 |
| 0.3086 0.6094 0.0820 0 0 |
|    0      0      0    1 0 |
```

Example 37-10 uses this matrix to desaturate an image.

EXAMPLE 37-10   http://actionscriptbible.com/ch37/ex10

Converting to Grayscale with a ColorMatrixFilter

```
package {
  import flash.display.Sprite;
  import flash.filters.ColorMatrixFilter;

  public class ch37ex10 extends Sprite {
    function ch37ex10() {
      var img:TestImage = new TestImage();
      addChild(img);
      var grayMat:Array = [0.3086, 0.6094, 0.082, 0, 0,
                           0.3086, 0.6094, 0.082, 0, 0,
                           0.3086, 0.6094, 0.082, 0, 0,
                                0,      0,     0, 1, 0];
      img.filters = [new ColorMatrixFilter(grayMat)];
    }
  }
}
import flash.display.Loader;
import flash.net.URLRequest;
import flash.system.LoaderContext;
class TestImage extends Loader {
```

EXAMPLE 37-10   *(continued)*

```
  public function TestImage(scale:Number = 1, x:Number = 0, y:Number = 0) {
    //photo   (CC-BY) Roger Braunstein
    //source  http://www.flickr.com/photos/rogerimp/2940373537/
    var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
    load(new URLRequest(url), new LoaderContext(true));
    scaleX = scaleY = scale;
    this.x = x; this.y = y;
  }
}
```

Saturation

Converting an image to grayscale is a special case of modifying the saturation of an image. Saturation measures the amount or vividness of color in an image; converting to grayscale is called desaturation because, of course, it removes all color from an image. But you can modify the saturation of an image with more accuracy than simply all-or-nothing.

Changing the saturation does two things simultaneously. It tones down each channel in much the same way that you scaled brightness (although this time you're going to use the luminance coefficients instead of giving each channel equal weight). At the same time, changing the saturation gives each channel an extra dose of its own color. In other words, it turns up the color of each channel while turning down the brightness overall to compensate. To calculate the red channel with a saturation scale factor of s, and the luminance coefficient constants 0.3086, 0.6094, and 0.0820 given by rw, gw, and bw, the equation is

```
    R' = ((1-s)rw + s) * R  + ((1-s)gw) * G  +  ((1-s)bw) * B
```

Every channel is scaled down as the saturation scales up, but the red channel is scaled up. In other words, the contribution of each color to its own channel is emphasized, enhancing the saturation of each color.

Example 37-11 changes the saturation of the image with the x position of the mouse.

EXAMPLE 37-11   http://actionscriptbible.com/ch37/ex11

Adjusting Saturation with a ColorMatrixFilter

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.filters.ColorMatrixFilter;

  public class ch37ex11 extends Sprite {
    protected var img:TestImage;
```

```
    function ch37ex11() {
      img = new TestImage();
      addChild(img);
      addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    }
    protected function onMouseMove(event:MouseEvent):void {
      //1 for no change, 0 for desaturate, >1 for high saturation
      //mouse at left for grayscale, right for 3x saturation boost
      var s:Number = stage.mouseX / stage.stageWidth * 3;
      var R:Number = (1 - s) * 0.3086;
      var G:Number = (1 - s) * 0.6094;
      var B:Number = (1 - s) * 0.0820;
      var satMatrix:Array = [R+s,   G,   B, 0, 0,
                             R, G+s,   B, 0, 0,
                             R,   G, B+s, 0, 0,
                             0,   0,   0, 1, 0];
      img.filters = [new ColorMatrixFilter(satMatrix)];
    }
  }
}
import flash.display.Loader;
import flash.net.URLRequest;
import flash.system.LoaderContext;
class TestImage extends Loader {
  public function TestImage(scale:Number = 1, x:Number = 0, y:Number = 0) {
    //photo   (CC-BY) Roger Braunstein
    //source  http://www.flickr.com/photos/rogerimp/2940373537/
    var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
    load(new URLRequest(url), new LoaderContext(true));
    scaleX = scaleY = scale;
    this.x = x; this.y = y;
  }
}
```

Convolution Filters

You can run convolution filters in Flash Player by using the `ConvolutionFilter` class. A convolution filter uses a pixel's neighbors to help determine its output value. The amount that each neighbor contributes to the output is defined by a convolution matrix. Rather than using matrix multiplication, this matrix simply provides the influence of the neighboring pixels. When calculating the output value of the filter on a given pixel, its neighbors are pulled from the image in the surrounding area, with the target pixel in the center. You can provide convolution matrices of different sizes; these matrices don't even need to be square. This 3×3 convolution matrix leaves an image unchanged, because the output value for each pixel is 1 times its original value (the value at the center of the matrix) and 0 times all its neighbors.

```
| 0 0 0 |
| 0 1 0 |
| 0 0 0 |
```

Now you'll look at applying a simple convolution matrix to a small image. The convolution matrix is the 3×3 matrix on the left, and the image is the 5×5 image on the right.

```
| a b c |    | 00 01 02 03 04 |
| d e f |    | 10 11 12 13 14 |
| g h i |    | 20 21 22 23 24 |
             | 30 31 32 33 34 |
             | 40 41 42 43 44 |
```

Given this setup, to determine the output of the pixel at (1,1), labeled 11, you would see the following contributions:

```
output(1, 1) = a*00 + b*01 + c*02
             + d*10 + e*11 + f*12
             + g*20 + h*21 + i*22
```

In fact, `ConvolutionFilter` uses this formula but with two changes. You can use a different divisor than 1 for the weighted average. By default, the divisor is 1, so all the coefficients in the convolution matrix should sum to 1 to keep colors in acceptable bounds. But by setting the `divisor` property, you can choose another number to divide the product above by.

The other factor omitted in this formula is a final offset value to add. This is determined by the value of the `bias` property of the `ConvolutionFilter`, which defaults to 0.

What happens when you need to calculate the value of the pixel at (4, 0), labeled 40? If you line up the center of the convolution matrix to that pixel, the values a, d, g, h, and i fall outside the image. `ConvolutionFilter` gives you two options. It can extend the colors at the edge of the image to "outside" the image. This is called *clamping*. In this case, the value i corresponds to the pixel labeled 41, even though you've already used that pixel with the f coefficient. You simply extend the edges of the image down and to the left (or in other directions, when you're processing a pixel on other edges of the image) as far as necessary to provide a corresponding value for every coefficient in the convolution matrix. The alternative is to use a predefined color value for all pixels outside the defined image.

You can choose whether to clamp the image or use a preset value with the `clamp` Boolean property of a `ConvolutionFilter`. The filter uses clamping by default. When set to `false`, you can provide the "out-of-bounds" color with the `color` and `alpha` properties.

You provide the matrix to this filter as a one-dimensional array with the matrix contents in row-major order, left to right and top to bottom. You tell the `ConvolutionFilter` what dimensions to interpret your matrix as with its `matrixX` and `matrixY` properties. For example, to define the following 3×5 matrix

```
| a b c d e |
| f g h i j |
| k l m n o |
```

you would use the following code:

```
var f:ConvolutionFilter = new ConvolutionFilter();
f.matrixX = 5;
f.matrixY = 3;
f.matrix = [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o];
```

It's important to set the dimensions of the input matrix before setting the matrix. You may also set all three at the same time using `ConvolutionFilter`'s constructor.

Here are the properties of `ConvolutionFilter`:

- `matrixX` — Number of columns in the convolution matrix.
- `matrixY` — Number of rows in the convolution matrix.
- `matrix` — Array defining the elements of the convolution matrix. Its size must correlate to the dimensions provided by `matrixX` and `matrixY`.
- `divisor` — Number that the sum of the pixel contributions is divided by.
- `bias` — Number added to the sum of the pixel contributions.
- `preserveAlpha` — Whether to apply the convolution to alpha values. When `true`, the result of the convolution affects only the red, green, and blue channels, and alpha values are unchanged. Defaults to `true`.
- `clamp` — Whether to clamp edges of the image. Defaults to `true`.
- `color` — If `clamp` is `false`, the out-of-bounds color as a `uint`.
- `alpha` — If `clamp` is `false`, the alpha of the out-of-bounds color.

Like other filters, these parameters may be provided to the constructor in this order, and all are optional. Convolution matrices are used in computer vision and computer graphics because they are fast and versatile. You can use different matrices to sharpen an image, blur it, detect and enhance edges, do cheesy embosses and bevels, or create other effects that operate on local features of the image.

Note

Under certain conditions, 3×3 convolution matrices are accelerated if your processor supports a certain extended instruction set. Check the AS3LR under the `matrix` property of `ConvolutionFilter` for details. ∎

Example 37-12 shows a few useful convolution matrices.

EXAMPLE 37-12   `http://actionscriptbible.com/ch37/ex12`

Multiple Convolution Matrices

```
package {
  import flash.display.Sprite;
  import flash.filters.ConvolutionFilter;
  public class ch37ex12 extends Sprite {
    function ch37ex12() {
      var a:TestImage = new TestImage(0.5, 0, 0);
      var b:TestImage = new TestImage(0.5, 250, 0);
      var c:TestImage = new TestImage(0.5, 0, 167);
      var d:TestImage = new TestImage(0.5, 250, 167);
      addChild(a);
      addChild(b);
      addChild(c);
      addChild(d);
```

EXAMPLE 37-12 *(continued)*

```
    var aMat:Array = [0,  1, 0,
                      1, -4, 1,
                      0,  1, 0]; //edge detect
    a.filters = [new ConvolutionFilter(3, 3, aMat)];
    var bMat:Array = [ 0, -1,  0,
                      -1,  5, -1,
                       0, -1,  0]; //sharpen
    b.filters = [new ConvolutionFilter(3, 3, bMat)];
    var cMat:Array = [1, 1, 1,
                      1, 1, 1,
                      1, 1, 1]; //blur
    c.filters = [new ConvolutionFilter(3, 3, cMat, 9)];
    var dMat:Array = [-2, -1, 0,
                      -1,  1, 1,
                       0,  1, 2]; //emboss
    d.filters = [new ConvolutionFilter(3, 3, dMat)];

  }
 }
}
import flash.display.Loader;
import flash.net.URLRequest;
import flash.system.LoaderContext;
import flash.events.MouseEvent;
import flash.geom.Point;
class TestImage extends Loader {
  protected var scale:Number;
  protected var origin:Point;
  public function TestImage(scale:Number = 1, x:Number = 0, y:Number = 0) {
    //photo   (CC-BY) Roger Braunstein
    //source  http://www.flickr.com/photos/rogerimp/2940373537/
    var url:String = "http://actionscriptbible.com/files/heiwadoori.jpg";
    load(new URLRequest(url), new LoaderContext(true));
    this.scale = scale;
    this.origin = new Point(x, y);
    addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    onMouseUp(null);
  }
  protected function onMouseDown(event:MouseEvent):void {
    parent.setChildIndex(this, parent.numChildren-1);
    scaleX = scaleY = 1;
    x = y = 0;
  }
  protected function onMouseUp(event:MouseEvent):void {
    scaleX = scaleY = scale;
    x = origin.x; y = origin.y;
  }
}
```

The first matrix, `aMat`, enhances an image so that edges are highlighted (usually in crazy neon colors.) Think a bit about how this is done. A pixel's four neighbors in every direction are added up, and the pixel is subtracted away. When the neighbors up, down, left, and right are close to the input pixel value, the four positive pixels and the −4x pixel should cancel out. But when the color changes abruptly in one or more of those directions, the difference between the sum and the center pixel is significant. So areas of little change appear black, and areas where the color changes abruptly in one of the four cardinal directions get colored. You can perform more simple edge detection biased for a single direction by balancing out the center pixel and its neighbor in one direction with opposite coefficients, setting all other elements to zero.

The `bMat` matrix works in almost the same way, except that the center pixel is weighted one more than the sum of the other pixels — so the original colors are preserved and added onto rather than subtracted away. Where there is little localized contrast, additional brightness is sucked away by the four −1x pixels on the sides, leaving at minimum the input value remaining. Where there is a lot of local contrast, the center pixel is multiplied in brightness: with less than a full −1x contribution coming from each of the four neighboring pixels, more of the center pixel's 5x coefficient remains. So pixels where the neighbors are of different brightness are enhanced, which appears to sharpen the image.

The `cMat` matrix produces a simple Gaussian blur, much like a `BlurFilter` can provide. A Gaussian blur averages neighboring pixels within a given radius. To blur more or less, you would create matrices of the same form with greater or smaller sizes. A larger matrix takes into account pixels farther away, thus increasing the radius or "blurriness" of the blur. Note that rather than set each coefficient to 1/9, I've written in all the coefficients as 1 and used a `divisor` of 9. This saved me some typing.

Finally, `dMat` produces a cheesy-looking emboss. By inversely removing contributions from the upper-left quadrant of neighbors and adding contributions from the bottom-right quadrant of neighbors, you emphasize pixels where the contrast is high in one direction, as if they are lit up by light coming from that direction — in this case, a diagonal from bottom right to top left.

So that you can look at the results in more detail, holding down the mouse button on any of the images scales the image up temporarily.

These filters, and others you can perform with a convolution filter, may not be high quality, but they are excellent for preparing images and video for analysis by computer vision algorithms. They run fast and can emphasize certain local features of images that may be important.

Displacement Maps

The `DisplacementMapFilter` class lets you arbitrarily distort a display object. Common applications for this include simulating an image projected on a surface, such as a map projected onto a globe, a label projected onto a cylinder, a panoramic photo projected into the inside of a sphere or cube, or the surface of a flapping banner; interactive distortions like Photoshop's Liquefy filter or Kai's Power Goo (if you remember that); optical distortions like an image viewed through glass or water; and distortions like pinching, bloating, and nonuniformly stretching.
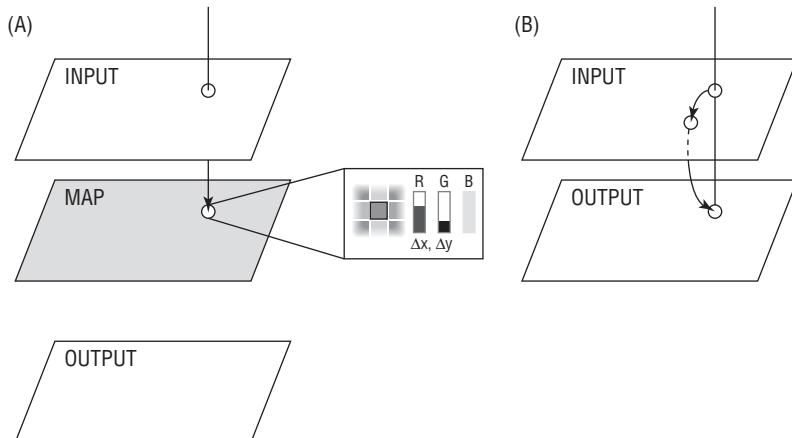
A displacement map filter doesn't change the colors or appearance of an image like the filters covered previously. Instead, it distorts the image, and it uses a displacement map, a second image, to define how the image should be distorted. To find out what to draw in pixel (x, y), the filter looks at the displacement map at (x, y) and uses its color to determine an offset. It adds that offset to (x, y) and looks at this new location for the pixel to copy into the output image. So, for example, if you have a

displacement map of a solid color, the result of the filter will be the original image, undistorted, offset a constant amount: Every location the filter looks at returns the same color, which maps into the same offset. So every output pixel gets the value of an input pixel a constant distance away. Figure 37-1 shows how this works for a single pixel.

Drawing one pixel with a displacement map applied

You use bitmap image data as a displacement map, whereas other filters you've seen in this book use a matrix. This is simply a matter of managing data. In previous filters, you've used relatively small matrices. To determine the offset of a pixel, you need to store two values for its offset in x and y. And you need an offset value for every pixel in the source image, so two matrices of size W×H are required, where the input image is W by H pixels. Dealing with, and generating or loading, these potentially huge matrices would be unwieldy. Instead, a bitmap image is used to store the data. Each pixel's color is a combination of red, green, blue, and alpha channels, and you exploit this to store multiple channels of data (the x and y offsets) in one image. You can store four 8-bit (1-byte) values in every position of a bitmap image in Flash Player if you consider each color channel independently. In a displacement map, only two are used. Each channel can store values in the range [0, 255], which are normalized (scaled and shifted) to the range [−0.5, 0.5]. This has one important implication: a black (0x00000000) pixel in the distortion map actually encodes the maximum negative displacement in both directions, not a zero-distance displacement. A 50 percent gray (0x80808080) pixel encodes this instead.

Say you have an input image `input`, and a displacement map image `map`, and you've decided to encode the x offsets in `map`'s red channel and the y offsets in its green channel. In pseudocode,

```
offset(x, y) = (normalize(map(x, y).red) * scaleX,
                normalize(map(x, y).green) * scaleY)
output(x, y) = input((x, y) + offset(x, y))
```

You can use an algorithm to generate a displacement map programmatically, using the bitmap drawing methods described in Chapter 36, "Programming Bitmap Graphics," or using vector drawing tools and

capturing the drawing in a `BitmapData` object with the `draw()` method. For projections defined by optics or geometry, it makes sense to generate these maps in code. You can also simply load an image as a `Bitmap` and use its `bitmapData`. No matter how the displacement map is created, you pass it to the `DisplacementMapFilter` as `BitmapData`.

Following are the filter's properties:

- `mapBitmap` — The displacement map.

- `mapPoint` — When the displacement map and input image are not aligned, a `Point` that describes the origin (top left) of the input with respect to the map.

- `componentX` — Which color channel from the map bitmap encodes the horizontal displacement. One of `BitmapDataChannel.RED`, `BitmapDataChannel.GREEN`, `BitmapDataChannel.BLUE`, or `BitmapDataChannel.ALPHA`.

- `componentY` — Which color channel from the map bitmap encodes the vertical displacement. See `componentX` for possible values.

- `scaleX` — The maximum absolute horizontal displacement.

- `scaleY` — The maximum absolute vertical displacement.

- `mode` — How to displace pixels along the edges. This filter suffers from a similar problem as a convolution filter: what should it do when the post-displacement location in the source image is out of bounds? The filter provides the following options. `DisplacementMapFilterMode.IGNORE` uses the undisplaced source pixel instead. `DisplacementMapFilterMode.WRAP` uses the pixel that would be present if the source image tiled infinitely. `DisplacementMapFilterMode.CLAMP` uses the nearest in-bounds pixel. `DisplacementMapFilterMode.COLOR` uses a predefined substitute color instead. The default is `WRAP`.

- `color` — The substitute color when `mode` is `DisplacementMapFilterMode.COLOR`.

- `alpha` — The alpha value of the substitute color when `mode` is `DisplacementMapFilterMode.COLOR`.

The parameters in the given order are also the parameters to `DisplacementMapFilter`'s constructor, where they are optional.

In Example 37-13, Perlin noise, covered in Chapter 36, is used to animate a flag flapping in the wind.

EXAMPLE 37-13   `http://actionscriptbible.com/ch37/ex13`

Using a Displacement Map

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.filters.DisplacementMapFilter;
  import flash.filters.DisplacementMapFilterMode;
  import flash.geom.*;
  import flash.net.URLRequest;
  [SWF(width="500",height="500",backgroundColor="#000000",frameRate="50")]
```

continued

EXAMPLE 37-13   (continued)

```
public class ch37ex13 extends Sprite {
  protected const FLAG_WIDTH:Number = 315, FLAG_HEIGHT:Number = 170;
  protected const WIND_SPEED:Number = -15;
  protected const DISPLACEMENT:Number = 20;
  protected const OCTAVES:int = 3;
  protected const SHIFT_COLORS:ColorTransform
      = new ColorTransform(2, 2, 2, 1, -128, -128, -128, 0);
  protected var flag:DisplayObject;
  protected var fadeOutWave:Shape;
  protected var map:BitmapData;
  protected var offsets:Array;
  protected var perlinSeed:Number;
  protected var displacementFilter:DisplacementMapFilter;
  public function ch37ex13() {
    var l:Loader = new Loader();
    l.load(new URLRequest("http://actionscriptbible.com/files/flag.swf"));
    l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoadComplete);
  }
  protected function onLoadComplete(event:Event):void {
    flag = LoaderInfo(event.target).content;
    flag.width = FLAG_WIDTH;
    flag.height = FLAG_HEIGHT;
    flag.x = 20;
    map = new BitmapData(FLAG_WIDTH, FLAG_HEIGHT, false, 0x808080);
    displacementFilter = new DisplacementMapFilter(map, new Point(),
        BitmapDataChannel.RED, BitmapDataChannel.RED,
        DISPLACEMENT*0.7, DISPLACEMENT,
        DisplacementMapFilterMode.COLOR);
    addChild(flag);
    flag.filters = [displacementFilter];
    perlinSeed = int(Math.random() * int.MAX_VALUE);
    offsets = new Array();
    for (var i:int = 0; i < OCTAVES; i++) offsets[i] = new Point();

    fadeOutWave = new Shape();
    var m:Matrix = new Matrix();
    m.createGradientBox(FLAG_WIDTH/2, FLAG_HEIGHT);
    fadeOutWave.graphics.beginGradientFill(GradientType.LINEAR,
        [0x808080, 0x808080], [1, 0], [0, 255], m);
    fadeOutWave.graphics.drawRect(0, 0, FLAG_WIDTH/2, FLAG_HEIGHT);
    fadeOutWave.graphics.endFill();

    var shadeBitmap:Bitmap = new Bitmap(map);
    shadeBitmap.height = FLAG_HEIGHT + DISPLACEMENT*2;
    shadeBitmap.width = FLAG_WIDTH + DISPLACEMENT;
    shadeBitmap.y = -DISPLACEMENT;
    addChild(shadeBitmap);
```

```
        shadeBitmap.blendMode = BlendMode.OVERLAY;
        shadeBitmap.alpha = 0.4;

        this.x = 50; this.y = 50;
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
        for (var i:int = 0; i < OCTAVES; i++) {
            offsets[i].x += Math.pow((1+i), 0.2) * WIND_SPEED;
        }
        map.perlinNoise(map.width, map.height * 6, OCTAVES,
            perlinSeed, false, true, 0, true, offsets);
        map.colorTransform(map.rect, SHIFT_COLORS);
        map.draw(fadeOutWave);
        displacementFilter.mapBitmap = map;
        flag.filters = [displacementFilter];
    }
  }
}
```

By using Perlin noise as a displacement map, you can simulate vertical and horizontal waves in the fabric of the flag. The displacement map pulls double duty to shade the flag for added realism. Another visual tweak blends the displacement toward a no-offset displacement as you approach the left side of the flag, which would be presumably tethered and less reactive to wind shear. Note the color transform applied to the bitmap. This shifts the ranges of the colors generated by the Perlin noise generator so that they are centered on 128.

Shaders

Pixel shaders can be applied to display objects with the `ShaderFilter` class. This filter consumes a `Shader` object, set in the constructor or assigned to its `shader` property, and runs the shader, setting its first input to the display object it is filtering. Thus, your shader kernel must accept at least one `image4` input.

Version
FP10. Shaders, and `ShaderFilters`, are supported in Flash Player 10 and later. ∎

Shaders give you full programmatic control over the pixels on-screen. If you have an effect in mind that you can't achieve with any of Flash Player's built-in filters, you can write a custom shader for the effect. Likewise, you can replicate any of the filters covered in this chapter with a shader. Shaders are covered in detail in Chapter 38.

Summary

- Filters let you apply bitmap effects to display objects.
- You can apply a filter to a display object by adding a `BitmapFilter` subclass to the `filters` array property that every `DisplayObject` possesses.
- Filter effects are cumulative and applied in the order they appear in the `filters` array.
- Filter objects are stored and retrieved by copy, so you must assign a new `filters` array to update the filters applied to a display object.
- Flash Player has a number of built-in filters that can be used for many purposes.

Writing Shaders with Pixel Bender

Pixel Bender, a cross-product toolkit for image processing, opens a whole new set of possibilities in Flash Player version 10 and above. You can use Pixel Bender to generate graphics, to filter display objects, to create custom blend modes, and even to process heaps of nonimage data. Pixel Bender uses its own language entirely separate from ActionScript 3.0, and it is compiled before being used in Flash Player. In this chapter, you'll learn about the ways Pixel Bender can enhance your Flash applications, how to load and use Pixel Bender programs in ActionScript, and how to program in the Pixel Bender kernel language.

Version

FP10. Only Flash Player 10 and up can execute Pixel Bender kernels. Future versions of Flash Player may add support for Pixel Bender features not currently available, such as dependent values, region functions, loops, arrays, variable vector indices, and so on. ■

Introducing Pixel Bender

Pixel Bender is a pixel shader (or *fragment shader*, sometimes) system developed for use across Adobe Creative Suite products. In theory, you can use the same shader as a filter in Photoshop, After Effects, and during runtime in Flash Player. A pixel shader is a short program whose output is the color of a given pixel. When an area affected by a pixel shader needs to be rendered, the system runs the program on each pixel that needs to appear on-screen, and it faithfully renders that pixel as the color returned by the program. In this way, the aggregate effect of the pixel shader is to color, or shade, the region. You can also bend Pixel Bender to your will and make it run on nonimage data. In the general case, a single Pixel Bender program is called a *kernel*.

Note

In reality, each product that uses Pixel Bender has some specific limitations that mean some shaders won't be entirely portable. This chapter focuses on the kinds of Pixel Bender programs that run in Flash Player. You'll see that, currently, Flash Player imposes some serious limitations on Pixel Bender kernels that make them best kept simple. ■

Pixel shaders aren't like the kinds of programs that you learn to write in the rest of this book. Like regular expressions, they are tools perfectly suited for their singular task. Regular expressions don't look anything like the rest of ActionScript, and they are executed in their own fashion, to efficiently and concisely manipulate strings. Similarly, Pixel Bender shaders use a language of their own, the Pixel Bender kernel language, and are executed in a manner completely unlike ActionScript code, all in the service of quickly generating graphics. To write a pixel shader, you have to think like a pixel shader.

The Case for Pixel Shaders

Pixel shaders were created to take advantage of increasingly powerful 3D graphics cards. Before pixel shaders, lighting calculations and texture mapping on the graphics card determined the appearance of a 3D model. The effects you can achieve relying on this pipeline are limiting. With pixel shaders, developers could send a short program to the graphics card and have it execute this program on every pixel with blinding speed. The appearance of a pixel-shaded triangle is completely open to creativity. Why pixel shaders though? Couldn't developers write that functionality directly into their programs? In theory, the main program could render every pixel of every output frame. But the rest of the 3D graphics pipeline was on the graphics card; moving the scene back to the CPU to shade each pixel and then back to the graphics card for display would hardly be efficient. The point of pixel shaders is to take advantage of graphics cards' strengths and speed.

It's important to understand this background. It explains shaders' existence, but also their properties. To shade a 640 × 480 area at 30 frames per second, a pixel shader kernel must run over 9 million times a second. It can do this because pixel shaders are

- Highly parallel — Graphics cards run many instances of the shader at the same time, drawing their outputs into the buffer as they complete. Kernels can depend on varying inputs, but they can't depend on other shaded pixels, which would create a requirement that some pixels are shaded before others.

- Optimized to work on GPU hardware — In practice, this means that vector and matrix operations can be carried out very fast. You'll see that the language reflects this, as many of its data types are vectors.

- Compiled and optimized — Pixel shader languages are low level and are compiled before being sent to the graphics card.

That said, Flash Player currently executes Pixel Bender code on the CPU, not the GPU. Even so, Pixel Bender kernels run almost as if they were on a graphics card. They are executed in parallel, using all available CPU cores and using their instruction sets optimized for vector processing, and they are run asynchronous to the ActionScript Virtual Machine. Drawing graphics with Pixel Bender kernels blows the equivalent ActionScript bitmap code (like `setPixel()`) out of the water in terms of performance, though the gains are most evident on multicore processors.

Caution

Pixel Bender kernels are by nature intensive and may bog down even recent computers. Use sparingly. ■

Version

Flash Player 10.0 through 10.1 run Pixel Bender kernels on the CPU regardless of the render path, on both mobile and desktop platforms. Future releases may move Pixel Bender kernels onto the graphics card when possible. ■

Thinking Like a Pixel Shader

Now that you understand a little bit about how Pixel Bender kernels are run, you can get comfortable with how they're structured. I'll show some Pixel Bender kernel language here, just to get you used to how it looks. I won't focus on syntax, data types, or functions of the language until the section "Pixel Bender Kernel Language."

A Pixel Bender kernel is, as I've said, a self-contained program. The program is run, in parallel, once for every output pixel, and the output pixels are reassembled into the output image. A kernel is very much like a class. It can have member variables and functions, but it must always have one main function, called `evaluatePixel()`, which assigns a result value to a pixel. This way, Pixel Bender knows how to run the kernel and what to do with the output. You specify this like so:

```
<languageVersion : 1.0;>
kernel FillColor
< namespace : "com.actionscriptbible";
  vendor : "ActionScript 3.0 Bible";
  version : 1;
  description : "Fills image with a single color";
>
{
  output pixel4 result;

  void evaluatePixel() {
    result = pixel4(1.0, 0.0, 0.0, 1.0);
  }
}
```

Note

Code inside angle brackets <> is metadata. Although this is necessary for Pixel Bender, I'll exclude it in examples in this section to focus on the implementation. I'll bring it back when I get into language details. ■

You can see that the `kernel` looks like a `class`. Rather than `return` a value, you mark that the variable `result` will hold the output value with the `output` keyword and assign it a value, in this case the color red. Each time this is run, it returns the red color, so it turns each pixel red.

This simple kernel has no inputs. You can have multiple inputs for a kernel, a single input, or even no inputs. Perhaps the most common application of a shader is to determine the appearance of a single image (or polygon). This kind of shader might have a single input image. Now, even though the kernel is executed for a single pixel, the input isn't limited to a single pixel of the image. The kernel has access to the full image so that it can, for example, flip an image across the diagonal:

```
kernel Mirror <...>
{
  input image4 source;
  output pixel4 result;
```

805

```
void evaluatePixel() {
  float2 location = outCoord();
  location.xy = location.yx;
  result = sample(source, location);
}
}
```

You can see that the input is an entire image, even though the output is a single pixel. Given an input image `source`, you find out which pixel you're supposed to be drawing. Instead of a flat color, you pick the color from another pixel in the source image, because you have access to the whole image. You find the location of the output pixel with `outCoord()`, switch its y and x coordinates, and then look up the color at the new location. You could have just as easily picked the color 30 pixels to the right, or you could have divided the x coordinate by 2 to stretch out the image horizontally.

Maybe you want to draw a circle. To do so, you should pick a color to draw the pixels that fall inside the circle, and another for pixels that fall outside the circle:

```
kernel Circle <...>
{
 input image4 source;
 output pixel4 result;
 const float2 circleCenter = float2(50.0, 50.0);
 const float circleRadius = 50.0;

 void evaluatePixel() {
  float2 location = outCoord();
  if (distance(location, circleCenter) < circleRadius) {
    result = pixel4(1.0, 1.0, 1.0, 1.0);
  } else {
    result = pixel4(0.0, 0.0, 0.0, 1.0);
  }
 }
}
```

Once you've specified the circle's center and radius, you test the output pixel location. If it falls inside the circle, you color it white. Otherwise, you color it black. The result is a white circle on a black background. This kernel ignores — in fact, replaces — the contents of the input image.

As I demonstrated with the previous two examples, when you write pixel shaders, you have to think about their effects *implicitly*, which might take some getting used to. Rather than setting an image's scale or procedurally drawing a circle, you have to tell a program what to do with each pixel. The simple rules, when applied to every pixel, result in a consistent output. Some filters ask you to think in the same way. The convolution filter asks, "How is the color of each pixel influenced by its neighbors?" And the displacement map filter asks, "How far away from the output pixel location should I sample the source image?" Of course, you can replicate these filters easily with the tools Pixel Bender provides to you.

Now you'll go back to the inputs to a kernel. You've seen shaders with zero and one image inputs. When would you use more image inputs? How about building a shader to be used as a blend mode? Blend modes determine how an image looks after being layered on top of another image. For example, the `BlendMode.MULTIPLY` mode multiplies the colors in the top image with the cumulative colors below it. A kernel that's going to be used as a blend mode should take two images and output a single image.

```
kernel MultiplyBlendMode <...>
{
  input image4 source1;
  input image4 source2;
  output pixel4 result;

  void evaluatePixel() {
    result = sample(source1, outCoord()) * sample(source2, outCoord());
  }
}
```

That was easy! For each output pixel, sample the corresponding pixels in the two layers and multiply their colors. You can easily implement all the built-in blend modes with a line or two of Pixel Bender code.

Besides the input images, Pixel Bender shaders can take parameters of any type. These parameters are available to Flash Player so that you can create interactive shaders that respond to input from Action-Script, whether that's user-directed input or the result of an animation or algorithm.

In summary, when you're developing pixel shaders, you ultimately have to answer the question, "What is the color of a single pixel?" not the question, "What does the effect look like?" To determine this, you can access input images and input parameters and do all sorts of math, as long as you output a single color. Flash Player relentlessly executes your program in parallel for every pixel in the output image, taking advantage of your CPU as much as it can.

Integrating Pixel Bender and Flash Player

Pixel Bender kernels may be run by Flash Player as a fill, a stroke fill, a filter, a blend mode, or simply as a job. In each application, the results of the kernel are used differently. But in every application, the kernel is encapsulated the same way.

A Pixel Bender kernel is represented in ActionScript 3.0 as an instance of the Shader class. This class acts as the interface between the two technologies. It owns the kernel code and proxies the parameters you define in the kernel. But it doesn't go further. Unlike regular expressions or E4X, the Pixel Bender kernel language is not part of the AS3 language spec. It can't be inlined with your existing code.

Flash Player consumes Pixel Bender kernels only after they have been compiled to bytecodes. So you need to feed your otherwise-empty Shader instance a compiled kernel. First compile the kernel for Flash Player by using either the Adobe Pixel Bender Toolkit's Export Filter for Flash Player menu or the command-line tool pbutil, also included with the Adobe Pixel Bender Toolkit. This results in a PBJ file, a Pixel Bender Kernel bytecode.

Note

A bytecode is a compiled, machine-independent intermediate representation of the code; a SWF contains ActionScript bytecodes. Flash Player actually takes Pixel Bender kernel bytecodes and compiles them on demand into machine code that runs right on your CPU. By compiling bytecode at runtime (JIT, or *Just In Time compilation*), you can optimize one centrally-distributed bytecode on the user's machine for the user's CPU, which differs in architectures and features. ∎

How you feed this bytecode file to Flash Player is up in the air. You've already learned several ways you can do so. The Shader instance exposes a byteCode property of type ByteArray through

which you can set the bytecode, or you can pass the data into the constructor. You learned about how to deal with binary data in Chapter 13, "Binary Data and ByteArrays."

Once the shader is running, it should determine the correct image inputs by its context; parameters to the shader, metadata, and inputs of the kernel are exposed in `Shader`'s `data` property, a dynamic object of type `ShaderData`.

The section "Interfacing with Pixel Bender Kernels" shows code to load, use, and twiddle with kernels in ActionScript.

Pixel Bender Kernel Language

You've already seen a few extremely simple kernels, during which I avoided talking about the Pixel Bender kernel language too much. Now you can jump in, explore what the language is capable of, and code it up!

Syntax

As you have doubtless realized by now, the kernel language is not like ActionScript 3.0. But it's not entirely foreign either. The Pixel Bender kernel language is akin to GLSL, OpenGL's shader language. Some key things to note:

- Types are placed before the variable name, without a colon, as in C.

  ```
  float radius = 1.0;
  ```

- In function declarations, the return value comes before the name of the function, and the parameters place the types before the parameter names. However, Pixel Bender in Flash Player does not currently allow you to call functions you define yourself.

  ```
  void evaluatePixel() {}
  float2 customFunction(float a, float b) {} //not in Flash Player!
  ```

- Functions are overloaded — that is, many functions may exist by the same name, with different arguments. The correct version will be called depending on its arguments. This makes for flexible handling of different dimensions. The `length()` function, for instance, calculates the length of a one, two, three, or four-dimensional vector, depending on the type of its argument:

  ```
  float length(float x) {}
  float length(float2 x) {}
  float length(float3 x) {}
  float length(float4 x) {}
  ```

- Literals are not automatically converted between numeric types. Include a decimal somewhere in your literal to indicate it is a `float` (a floating-point number like `Number`). Leave out the decimal to interpret it as an `int`.

  ```
  float a = 2.0; //Preferred
  float b = .4;
  float c = 5.;
  float d = 6; //ERROR
  int e = 7;
  ```

Knowing this, you should be able to find your way around a Pixel Bender kernel. These programs are structurally simple, so there are no classes or OOP, and only a few built-in types and functions. The Adobe Pixel Bender Toolkit is an IDE for these programs, so its editor gives you code hinting and compiler errors and warnings. Be sure to turn on Flash Player warnings and errors in the IDE to limit yourself to Pixel Bender features supported in Flash Player.

Tip

If you want to create a kernel that executes specific code only if the kernel is being used in Flash Player, use the compiler preprocessor symbol `AIF_FLASH_TARGET`. This is a C-style `#define`, so you can include code if the kernel is being run in Flash Player. Instead of `#ifdef`, use `#if`. There is no `#ifndef`.

```
#if AIF_FLASH_TARGET
  //code that runs if you're in Flash Player
#endif
//code that runs no matter what
#if !AIF_FLASH_TARGET
  //code that runs if you're NOT in Flash Player
#endif
```

Structure of a Kernel

A Pixel Bender kernel is like a simplified class. Every kernel compatible with Flash Player follows this template:

```
<languageVersion: 1.0;>
kernel (kernel name)
<
  namespace: (a namespace);
  vendor: (name of your entity);
  version: (version number);
>
{
  (member declarations including consts, parameters, inputs, and outputs)
  void evaluatePixel() {
    (statements, including an assignment to the output)
  }
}
```

I've removed all elements that Flash Player doesn't support from this template. Most importantly, no extra functions are allowed in the kernel. Pixel Bender defines several other reserved functions, but they all relate to Pixel Bender operations that Flash Player does not currently support, like regions and dependents. Your only function should be `evaluatePixel()`, so I've baked it into the template.

The metadata included in this template is also required. Metadata is placed within angled brackets and is in pairs of

```
name: value;
```

The semicolons are required, including the one trailing the last name-value pair in the metadata block.

The name of the kernel is pretty arbitrary. Name it like you would name a class. Don't use periods in the name: a namespace, if required, can go in the kernel metadata.

Required Metadata

The language version tag simply marks the source code as conforming to a certain version of the Pixel Bender kernel language. For now, 1.0 is the only version, so just use this.

The kernel metadata, also required, includes the three pairs you see in the template and may contain an optional `description`. The tags should be self-explanatory. See the example:

```
kernel FillColor
< namespace : "com.actionscriptbible";
  vendor : "ActionScript 3.0 Bible";
  version : 1;
  description : "Fills image with a single color";
>
```

Tip

The Pixel Bender Toolkit generates a working template for you when you choose New Kernel, automatically filling in your `vendor` and `namespace` from its preferences. ■

Member Declarations

The member declarations are like instance variables in a class, but each one must have a stated purpose. You can only declare members that are constants, parameters, inputs, or outputs. For a basic instance variable, instead use a local variable declaration inside `evaluatePixel()`.

All member declarations take the form:

```
(member type) (data type) (name) [= (literal value for const members)];
```

For example:

```
const float circleRadius = 25.0;
input image4 inputImage;
output pixel4 returnPixel;
```

A constant should need no explanation. Flash Player usually assigns the inputs and outputs of a kernel automatically. That's where parameters come in.

Parameters make up the public interface to the kernel. They may be modified by ActionScript code while the kernel is running, and you can tweak them by hand to test your kernel if you're using the IDE. You could convert the constant values that define a circle in your `Circle` kernel from earlier into parameters:

```
<languageVersion: 1.0;>
kernel Circle <namespace: "as3b"; vendor: ""; version: 2;>
{
  input image4 source;
  output pixel4 result;
  parameter float2 circleCenter <
    minValue: float2(0.0, 0.0);
    maxValue: float2(1000.0, 1000.0);
    description: "Center point of the circle";
  >;
```

```
parameter float circleRadius <
  minValue: 1.0;
  maxValue: 500.0;
  defaultValue: 25.0;
>;

void evaluatePixel() {
  float2 location = outCoord();
  if (distance(location, circleCenter) < circleRadius) {
    result = sample(source, location);
  } else {
    result = pixel4(0.0, 0.0, 0.0, 0.0);
  }
}
}
```

When you use parameters, you can optionally add metadata to the parameters that set a minimum, maximum, and default value. All these values are optional. If you define none of these, you may omit the whole metadata block.

If you run this example in the IDE, you'll see that sliders appear to control the parameter values, and these sliders respect your defined ranges. You might also notice that I've changed the effect slightly. Instead of drawing a white circle inside a black field, it draws a spotlight on the image — a circular mask. If the pixel is inside the circle, you sample the source image. If the pixel is outside the circle, you return not just black but a completely transparent color.

Basic Control Structures

The only kernel language control structures that Flash Player supports are `if` and `if/else`. No loops, no nothing! This might seriously quash some ideas you might have had for Pixel Bender kernels, like maybe if you're trying to come up with a really cool example to write in a book, so think ahead and determine whether Pixel Bender is the right application.

Types

The basic types in the Pixel Bender kernel language are `bool`, `int`, `float`, and `pixel1`.

- `bool` — A Boolean value that takes the keywords `true` and `false`.
- `int` — An integer value, signed. In Flash Player it is 16 bits.
- `float` — A floating-point value. In Flash Player it is 32 bits.
- `pixel1` — A single channel of a pixel. Flash Player uses 4-channel RGBA pixels. `pixel1` is one 32-bit floating-point channel; it's identical to a `float`.

Caution

Flash Player stores bitmaps as 4 channels, ordered ARGB, 8 bits per channel as an unsigned integer from 0 to 255. Pixel Bender uses 4 channels, ordered RGBA, 32 bits per channel as a floating-point number from 0.0 to 1.0. Conversion in both directions is automatic. ■

Vector Types

Each of these types has vector versions, which wrap several values into a single variable.

- `bool2`, `bool3`, `bool4` — Vectors of Booleans.
- `int2`, `int3`, `int4` — Vectors of integers.
- `float2`, `float3`, `float4` — Vectors of floating-point numbers. Useful for positions in 2D and 3D space, quaternions, and so on.
- `pixel2`, `pixel3`, `pixel4` — Vectors of multichannel pixels. Most often, you'll use `pixel3`, which can store RGB channels, and `pixel4`, which can store RGBA channels.

You can create vector literals by calling the "constructor" function with the correct number of arguments or with a single argument, which fills all the elements with that value.

```
float3 center = float3(0.0, 0.0, 0.0);
float3 alsoCenter = float3(0.0);
bool2 options = bool2(true, false);
```

Vector types are awesome because operators are overloaded to work on vector types. This means that you can add two `float4`s in precisely the same way that you add two `float`s. In the `MultiplyBlendMode` shader example, you simply multiplied two pixel values:

```
result = sample(source1, outCoord()) * sample(source2, outCoord());
```

Each call to `sample()` here returned a `pixel4` because the images were 4-channel images. So with one line, four multiplications and four assignments were performed. Arithmetic operators work on each element of a vector independently. The dot product and cross product can be performed with functions. If you weren't able to use vector multiplication, you would have had to use this equivalent but tedious (and typo-prone) code:

```
pixel4 a = sample(source1, outCoord());
pixel4 b = sample(source2, outCoord());
result.r = a.r * b.r;
result.g = a.g * b.g;
result.b = a.b * b.b;
result.a = a.a * b.a;
```

Another cool thing about vectors in the Pixel Bender kernel language is how you can access their elements. You can use vectors like ActionScript vectors and arrays, with bracket notation (`a[0]`, `a[1]`, `a[2]`, `a[3]`). The kernel language gives you shortcuts — syntactic sugar — for these vector indices, based on the typical ways you'd use the vectors. You just saw one. Table 38-1 shows all of them. Each row in this table is an equivalent way to access the element.

These shortcuts are for your own convenience. Pixel Bender doesn't know or care what kind of data you're storing in a `float3`, as long as it's floating-point numbers. You use RGB or XYZ or STP to access its elements for one reason: to make the code more readable.

TABLE 38-1

Vector Index Shortcuts

Array Style	Color Channels	Coordinates	Texture Coordinates
[0]	.r	.x	.s
[1]	.g	.y	.t
[2]	.b	.z	.p
[3]	.a	.w	.q

Finally, vectors in the kernel language can be *swizzled*. This means that, using the shortcuts in Table 38-1, you can whisk around elements out of vectors and recombine them by using the index shortcuts, totally freeform. Try these:

```
float4 pxl = sampleNearest(src, outCoord());
float3 dropAlpha = pxl.rgb;
float4 grayFromRed = pxl.rrra;
float4 reversed = pxl.abgr;
```

Crazy, right?

Matrix Types

The kernel language also has built-in floating-point matrix types for these dimensions:

- `float2x2`, `float3x3`, `float4x4` — Square matrices of floating-point values, dimensions 2×2, 3×3, and 4×4.

These may be instantiated with the proper constructors, passing a list of column vectors, the proper total number of elements in column-major order, or with a single float that fills in the diagonal, leaving all other elements 0.

```
//define the 2x2 matrix:
// | 1.0 2.0 |
// | 3.0 4.0 |

float2x2 m = float2x2(1.0, 3.0, 2.0, 4.0);

float2 col0 = float2(1.0, 3.0);
float2 col1 = float2(2.0, 4.0);
float2x2 m = float2x2(col0, col1);
```

You can find an element with double bracket notation, column then row, or pull out a column vector with a single pair of brackets.

```
//pull out element 2.0
float element = m[1][0];

//pull out column vector <1.0, 3.0>
float2 firstCol = m[0];
```

The arithmetic operators are overloaded for matrices, too. In this case, multiplication of matrices with the same dimension actually performs matrix multiplication, not per-element multiplication. Multiplication of vectors and matrices with the proper dimension in the proper order also performs matrix multiplication. For example,

```
float2x2 * float2x2 -> float2x2
//multiplies 2x2 matrix by 2x2 matrix, yielding 2x2 matrix

float2 * float2x2 -> float2
//multiplies 1x2 matrix by 2x2 matrix, yielding 1x2 matrix (row vector)

float2x2 * float2 -> float2
//multiplies 2x2 matrix by 2x1 matrix, yielding 2x1 matrix (column vector)

float * float2x2 -> float2x2
//multiplies scalar by 2x2 matrix, yielding 2x2 matrix
```

Manipulating whole vectors and matrices at a time is not only concise and fast, but applicable to 3D graphics and some image processing algorithms.

Image Types

Suitable only for input images, Pixel Bender includes image types for images with various numbers of color channels.

- `image1, image2, image3, image4` — An image with the specified number of channels.

As I've already noted, you will be working mostly with 4-channel images, so you can get used to `image4`.

Pretty much the only thing you can do with an image is sample it, using the three sample functions listed toward the end of Table 38-2. This yields a pixel of the corresponding number of channels.

Stuff You Don't Get

Unfortunately, Pixel Bender on Flash doesn't support arrays. Instead, you could try to encode a series of float values in an n×1 one-channel image (`image1`). Regions are not yet available.

Functions

Pixel Bender provides a suite of mathematical functions, all of which are overloaded to work with vectors of any dimension when it makes sense. For example, `log()` can take the logarithm of a number if passed a `float`, but it also takes the logarithm of four numbers at once if passed a `float4`. I signify that any dimension of float may be passed with the (purely notational) type `floatn`. So few functions are defined that they can be summarized in Table 38-2.

TABLE 38-2

Pixel Bender Functions

Function Signature	What It Does
`floatn radians(floatn d)`	Converts d degrees to radians
`floatn degrees(floatn r)`	Converts r radians to degrees
`floatn sin(floatn r)`	Sine; r is in radians
`floatn cos(floatn r)`	Cosine
`floatn tan(floatn r)`	Tangent
`floatn asin(floatn x)`	Arcsine
`floatn acos(floatn x)`	Arccosine
`floatn atan(floatn slope)`	Arctangent
`floatn atan(floatn y, floatn x)`	Arctangent
`floatn pow(floatn a, floatn b)`	Raises a^b
`floatn exp(floatn x)`	Raises e^x
`floatn exp2(floatn x)`	Raises 2^x
`floatn log(floatn x)`	Natural logarithm of x
`floatn log2(floatn x)`	Base 2 logarithm of x
`floatn sqrt(floatn x)`	Positive square root of x
`floatn inverseSqrt(floatn x)`	Reciprocal of the positive square root of x
`floatn abs(floatn x)`	Absolute value of x
`floatn sign(floatn x)`	Sign of x. Returns -1.0, 0.0, or 1.0
`floatn floor(floatn x)`	Rounds down x to nearest whole number
`floatn ceil (floatn x)`	Rounds up x to nearest whole number
`floatn fract(floatn x)`	Fractional part of x
`floatn mod(floatn a, floatn b)`	Returns a modulo b, remainder of a divided by b
`floatn min(floatn a, floatn b)`	Minimum of the two arguments
`floatn max(floatn a, floatn b)`	Maximum of the two arguments

TABLE 38-2 *(continued)*	
Function Signature	**What It Does**
`floatn step(floatn a, floatn b)`	Returns 0.0 if the first argument is greater, 1.0 if the second argument is greater or they are equal
`floatn clamp(floatn x, floatn min, floatn max)`	Clamps a value `x` to the range [min, max] inclusive
`floatn mix(floatn a, floatn b, floatn t)`	Linear interpolation between `a` and `b`, with `t` in the range `0.0` (returns `a`) to `1.0` (returns `b`)
`floatn smoothStep(floatn edge0, floatn edge1, floatn x)`	Hermite interpolation for `x` between `edge0` and `edge1`, clamped to the range [edge0, edge1]
`float length(floatn x)`	Length/magnitude of vector `x`
`float distance(floatn x, floatn y)`	Distance between vectors `x` and `y`
`float dot(floatn x, floatn y)`	Dot product of vectors `x` and `y`
`float3 cross(float3 x, float3 y)`	Cross product of vectors x and y
`floatn normalize(floatn x)`	Normalizes `x` so that its magnitude is `1.0`
`floatnxn matrixCompMult (floatnxn x, floatnxn y)`	Per-element multiplication of matrices `x` and `y` where * would perform matrix multiplication
`booln lessThan(intn/floatn x, intn/floatn y)`	Compares elements of vector with <
`booln lessThanEqual(intn/floatn x, intn/floatn y)`	Compares elements of vector with <=
`booln greaterThan(intn/floatn x, intn/floatn y)`	Compares elements of vector with >
`booln greaterThanEqual (intn/floatn x, intn/floatn y)`	Compares elements of vector with >=
`booln equal(intn/floatn/booln x, intn/floatn/booln y)`	Compares elements of vector with ==
`booln notEqual(intn/floatn/booln x, intn/floatn/booln y)`	Compares elements of vector with !=
`bool any(booln x)`	Returns `true` if any element of `x` is `true`
`bool all(booln x)`	Returns `true` if all elements of `x` are `true`
`booln not(booln x)`	Negates all elements of vector
`pixeln sample(imagen img, float2 p)`	Returns pixel color of image `img` at location `p`, using bilinear interpolation between pixels

Function Signature	What It Does
`pixeln sampleLinear(imagen img, float2 p)`	Returns pixel color of image `img` at location `p`, using bilinear interpolation between pixels
`pixeln sampleNearest(imagen img, float2 p)`	Returns pixel color of image `img` at location `p`, using nearest-neighbor to round position
`float2 outCoord()`	Returns the pixel location that this instance of the kernel is currently being run on
`float2 pixelSize(imagen/pixeln)`	Returns the size of a single pixel (*not* the size of the image) for an input image or output pixel; Flash Player returns `<1.0, 1.0>`
`float pixelAspectRatio(imagen/pixeln)`	Returns the aspect ratio of a pixel (for non-square pixels) for an input image or output pixel; Flash Player returns `1.0`

And there you have, in a nutshell, everything Pixel Bender can do. The rest is up to your imagination!

Interfacing with Pixel Bender Kernels

In this section, I'll show how to load in a kernel's bytecode and get a `Shader` handle to the kernel. Once you have a `Shader`, I'll show the different ways to run it to generate or modify graphics. Then, with the kernel running, I'll show ways to interface with it.

Loading the Bytecode

Recall from the section "Introducing Pixel Bender" that you always load compiled Pixel Bender bytecodes, typically stored as PBJ files. Any method of loading binary data will suffice.

You can load in the PBJ file from the web using `URLLoader`, as you saw in Chapter 27, "Flash Player Networking Basics."

```
var urlRequest:URLRequest = new URLRequest("shader.pbj");
var urlLoader:URLLoader = new URLLoader();
urlLoader.dataFormat = URLLoaderDataFormat.BINARY;
urlLoader.addEventListener(Event.COMPLETE, createShader);
urlLoader.load(urlRequest);

function createShader(event:Event):void {
  var shader:Shader = new Shader(ByteArray(event.target.data));
}
```

You can embed the bytecode in the SWF during compilation with an `Embed` metadata tag, as you did with fonts in Chapter 17, "Text, Styles, and Fonts."

817

```
[Embed(source="shader.pbj", mimeType="application/octet-stream")]
var ShaderBytecode:Class;

var shader:Shader = new Shader(new ShaderBytecode());
```

Ensure that you use the `application/octet-stream` MIME type to embed the file as raw binary data. It will come out as a subclass of `ByteArray`.

You can even load a local PBJ file at runtime with `FileReference`, as you saw in Chapter 30, "File Access."

```
var fileReference:FileReference = new FileReference();
fileReference.addEventListener(Event.SELECT, onFileSelect);
fileReference.browse([new FileFilter("Pixel Bender Bytecode", "*.pbj")]);

protected function onFileSelect(event:Event):void {
  var fileReference:FileReference = FileReference(event.target);
  fileReference.addEventListener(Event.COMPLETE, onLoadComplete);
  fileReference.load();
}

protected function onLoadComplete(event:Event):void {
  var fileReference:FileReference = FileReference(event.target);
  new Shader(fileReference.data);
}
```

Tip

Although Flash Player loads only compiled Pixel Bender bytecodes, there's nothing technically preventing you from creating a valid kernel bytecode file in memory. James Ward has published a Flash Player PBJ runtime assembler based on the Haxe PBJ assembler. It is called pbjAS, and you can find it at `http://bit.ly/pbjas`. ∎

Running a Kernel in ActionScript

Now that you have a `Shader` instance, you can use it in several different contexts.

- As a fill style — Call `beginShaderFill()` instead of `beginFill()` on a `Graphics` object.

  ```
  shape.graphics.clear();
  shape.graphics.beginShaderFill(aShader);
  shape.graphics.drawCircle(0, 0, 200);
  ```

- As a stroke style — I first showed this in Chapter 35, "Programming Vector Graphics." Call `lineShaderStyle()` on a `Graphics` object.

  ```
  shape.graphics.lineStyle(20, 0, 1);
  shape.graphics.lineShaderStyle(aShader);
  shape.graphics.moveTo(0, 0);
  shape.graphics.lineTo(200, 200);
  ```

- As a filter — Simply wrap the `Shader` object in a `ShaderFilter` and use as any other `BitmapFilter`. The shader should take one image input.

  ```
  sprite.filters = [new ShaderFilter(aShader)];
  ```

- As a blend mode — This requires two steps. On the display object you want to blend with a custom shader, set the shader blend mode, and then assign the shader to the `blendShader` property. The shader should take two image inputs.

```
sprite.blendMode = BlendMode.SHADER;
sprite.blendShader = aShader;
```

- As a standalone job — Use the `ShaderJob` class to control the inputs, outputs, and execution of the kernel. This manner of execution enables some clever programming and warrants its own section. I'll show what standalone shader jobs can achieve in the later section titled "Bending Other Data."

With an effect in mind, it should be fairly clear which method you should use to generate the correct kind of graphics. If you're writing your own 3D engine and using a pixel shader to shade polygons, you'll probably use the kernel as a fill mode with `Graphics`'s `drawTriangles()` method. If you're generating graphics to fill a surface, you'd probably again use the kernel as a fill style and fill the whole surface with `drawRect()`. If you're modifying existing graphics, it usually makes sense to use a `ShaderFilter`. And so on.

Manipulating a Kernel

The real fun comes when you use ActionScript to interface with a `Shader` in real time. For example, say your shader is doing light calculations. You're going to need to send the `Shader` in the correct parameters for the angle of incidence and the strength of the light for it to actually work correctly in context. Otherwise, every surface would be shaded as if it had the same orientation and position relative to the light.

The data associated with a Pixel Bender kernel is made available to ActionScript in the `Shader`'s `data` property. This is a dynamic object of type `ShaderData`, whose properties can be these types:

- `String` — Metadata properties of the kernel itself are stored as name-value pairs in `ShaderData`.

```
trace(aShader.data.name); //Circle
trace(aShader.data.version); //2
```

- `ShaderInput` — Input images to the kernel are stored by their variable name. `ShaderInput` provides facilities for sending in "image" data as a `BitmapData`, a `ByteArray`, or a `Vector.<Number>`. You can use these methods to coerce Pixel Bender to crunch huge sets of data masquerading as images, without doing too much work on the ActionScript side to make them look like images. More on this later. In a filter context and a blend mode context, Flash Player can set one or two of the inputs automatically. If your kernel uses additional inputs, use `ShaderInput` to specify them.

```
trace(aShader.data.optionalImage.channels);
//4 if optionalImage is an image4, 3 if it's image3, etc.
aShader.data.optionalImage.input = someBitmapData;
```

- `ShaderParameter` — Parameters to the kernel are stored by their variable name, as instances of `ShaderParameter`. The `ShaderParameter` object gives you access to the metadata associated with the parameter, like `description`, `minValue`, `maxValue`, and `defaultValue`; its `type`; and its current `value`. For Flash Player, all values are converted into `Arrays` to support vector and matrix values. Even scalars are converted into `Arrays` of size 1.

```
              var circleCenter:ShaderParameter = shader.data.circleCenter;
              trace(circleCenter.type); //float2
              trace(circleCenter.minValue, "->", circleCenter.maxValue);
              //0,0 -> 1000,1000
              circleCenter.value = [10, 10]; //you can assign a value
```

Caution

Don't forget that you have to assign the value of a parameter to the `ShaderParameter`'s `value` property. A statement like

```
              aShader.data.aParameter = [10]; //wrong
```

is actually valid, since the `ShaderData` is dynamic and `aParameter` is not type checked. Furthermore, the parameter in your shader will take on the default value when you type this, so this kind of bug is terrifically annoying. Remember to use `value`!

```
              aShader.data.aParameter.value = [10]; //right
```

Through the metadata associated with a shader, you can discover its interface at runtime, and through its `ShaderParameters` and `ShaderInputs`, you can interface with the kernel at runtime.

A Basic Shader in ActionScript

In this example, you take the `Circle` shader version 2, which masks its content with a circle like a spotlight, and hook it up to ActionScript, as shown in Example 38-1. You can find the Pixel Bender kernel language source earlier in this chapter. You use the mouse's position to move the spotlight, and you resize the spotlight with the mouse wheel. It's been set up to mask your webcam feed.

EXAMPLE 38-1 http://actionscriptbible.com/ch38/ex1

Using a Shader as a Filter

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.filters.*;
  import flash.media.*;
  import flash.net.*;
  import flash.utils.ByteArray;

  public class ch38ex1 extends Sprite {
    protected var camVideo:Video;
    protected var shader:Shader;
    protected var shaderFilter:ShaderFilter;
    public function ch38ex1() {
      camVideo = new Video(stage.stageWidth, stage.stageHeight);
      camVideo.attachCamera(Camera.getCamera());
      addChild(camVideo);
```

```
      var PBJURL:String = "http://actionscriptbible.com/files/circle2.pbj";
      var loader:URLLoader = new URLLoader(new URLRequest(PBJURL));
      loader.dataFormat = URLLoaderDataFormat.BINARY;
      loader.addEventListener(Event.COMPLETE, onBytecodeLoaded);
    }
    protected function onBytecodeLoaded(event:Event):void {
      var loader:URLLoader = URLLoader(event.target);
      loader.removeEventListener(Event.COMPLETE, onBytecodeLoaded);
      shader = new Shader(ByteArray(loader.data));
      shaderFilter = new ShaderFilter(shader);
      stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
      stage.addEventListener(MouseEvent.MOUSE_WHEEL, onMouseWheel)
    }
    protected function onEnterFrame(event:Event):void {
      shader.data.circleCenter.value = [stage.mouseX, stage.mouseY];
      camVideo.filters = [shaderFilter]; //update the filter
    }
    protected function onMouseWheel(event:MouseEvent):void {
      shader.data.circleRadius.value[0] += 3 * event.delta;
    }
  }
}
```

The example explores all three topics covered in this section. You've loaded the bytecode into a Shader instance, in this case using a URLLoader. You've executed the shader, in this case as a filter for video content. And you've interfaced with its parameters to make the shader interactive.

This is about the simplest shader and application I could think of. It's kind of boring. Next I'll shoot for something cool. What's the point of Pixel Bender if you use it to re-create graphics you could make with existing drawing and bitmap APIs?

Tip

Okay, here's the point. Although boring, re-creating existing Flash Player effects can be instructional. Every built-in blend mode and BitmapFilter can be easily accomplished as a Pixel Bender shader. It would be an interesting exercise to compare performance of built-in effects with your Pixel Bender implementations. Given that kernels can execute on multiple cores, you may be able to beat Flash Player at its own game. ∎

An Effect Shader in ActionScript

Now you'll put together a complex effect in Pixel Bender. Within a single filter kernel, you can combine a bunch of different operations so that all the parts of the compound effect render at the same time. I've put together a "bad reception" Pixel Bender kernel that combines vertical roll, sinusoidal distortion, noise distortion, color channel splitting, and some horizontal lines.

This example uses some workarounds that might be helpful to any Pixel Bender developer. Because Pixel Bender doesn't have noise — and you can't loop to generate any — you use bitmap data and feed in the noise as an extra input. In general, when Pixel Bender can't generate the kind of input you need, try sending it in as image data.

Also, you pass in the dimensions of the image as parameter `dimensions`, because Pixel Bender can't actually tell you the size of images.

```
<languageVersion : 1.0;>
kernel BadReception
< namespace : "com.actionscriptbible";
  vendor : "ActionScript 3.0 Bible";
  version : 1;
  description : "Bad reception on your TV";
>
{
  input image4 srcImage;
  input image4 noiseImage;
  output pixel4 dst;

  parameter float2 dimensions
    <minValue: float2(0.0, 0.0); defaultValue: float2(640.0, 480.0);>;
  parameter float vRoll
    <minValue: -200.0; defaultValue: 0.0; maxValue: 200.0;>;
  parameter float channelSplit
    <minValue: 0.0; maxValue: 50.0;>;
  parameter float noisyHDisplace
    <minValue: -100.0; maxValue: 100.0;>;
  parameter float sinHDisplace
    <minValue: -100.0; maxValue: 100.0;>;
  parameter float3 sinHDisplaceFactors
    <minValue: float3(0.0); maxValue: float3(10.0);>;
  parameter float3 sinHDisplaceOffsets
    <minValue: float3(-6.0); maxValue: float3(6.0);>;
  parameter float noiseLayer
    <minValue: 0.0; maxValue: 1.0;>;
  parameter float blackoutThresh
    <minValue: 0.0; maxValue: 1.0;>;

  void evaluatePixel() {
    float2 coord = outCoord();
    //Vertical roll
    coord.y = mod(coord.y - vRoll, dimensions.y);
    //Displacements
    float2 displace = float2(0.0);
    displace.x = noisyHDisplace
      * (length(sampleNearest(noiseImage, float2(0.0, outCoord().y))) - 0.5);
    float3 sinDisplaces = sinHDisplace * sin(
      float3(coord.y / 100.0)
      * pow(sinHDisplaceFactors, float3(1.0, 2.0, 3.0))
      + sinHDisplaceOffsets
      );
    displace.x += sinDisplaces[0] + sinDisplaces[1] + sinDisplaces[2];
    coord += displace;
    //Color channel splitting
    if (channelSplit == 0.0) {
      dst = sampleNearest(srcImage, coord);
    } else {
```

```
          float2 channelDisplace = float2(0.0, 0.0);
          channelDisplace.x = -channelSplit;
          dst.r = sampleNearest(srcImage, coord+channelDisplace).r;
          channelDisplace.x = 0.0;
          dst.g = sampleNearest(srcImage, coord+channelDisplace).g;
          channelDisplace.x = channelSplit;
          dst.b = sampleNearest(srcImage, coord+channelDisplace).b;
        }
        //Black lines at top and bottom
        float blackoutPixel = pow(outCoord().y / dimensions.y * 2.0 - 1.0, 2.0);
        if (sampleNearest(noiseImage, float2(dimensions.x*0.5, outCoord().y)).r
          < blackoutThresh * blackoutPixel) {
            dst = sampleNearest(noiseImage, outCoord()) * 0.05;
        }
        //Noise
        dst -= noiseLayer * sampleNearest(noiseImage, outCoord());
        dst.a = 1.0;
      }
    }
```

The pieces of this effect are pretty simple by themselves, but they all come together. If you used `BitmapData` pixel operations and looped over each pixel in ActionScript, this effect would have trouble keeping up any reasonable frame rate. When the kernel is run in the Pixel Bender toolkit on the GPU, it's blinding fast. (Too bad you can't take advantage of that in Flash Player.)

Developing the filter is just the first step. To really sell the effect, you have to animate its parameters so that it really looks like TV interference. Example 38-2 shows the ActionScript code used to do that.

EXAMPLE 38-2 `http://actionscriptbible.com/ch38/ex2`

Using a Complex Shader

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.filters.ShaderFilter;
  import flash.geom.Point;
  import flash.media.*;
  import flash.net.*;
  import flash.utils.getTimer;
  [SWF(width="500",height="500",frameRate="30",backgroundColor="#000000")]
  public class ch38ex2 extends Sprite {
    protected var holder:Sprite;
    protected var video:Video;
    protected var testPattern:Loader;
    protected var noise:BitmapData;

    protected var shader:Shader;
    protected var shaderFilter:ShaderFilter;
```

continued

EXAMPLE 38-2   *(continued)*

```
  protected var smoothRandomNoise:BitmapData;
  protected var pointers:Vector.<Point>;
  protected var rands:Vector.<Number>;

  public function ch38ex2() {
    noise = new BitmapData(stage.stageWidth, stage.stageHeight, false, 0);
    holder = new Sprite();
    video = new Video(stage.stageWidth/2, stage.stageHeight/2);
    video.scaleX = video.scaleY = 2;
    var camera:Camera = Camera.getCamera();
    camera.setMode(stage.stageWidth/2, stage.stageWidth/2, 30, false);
    video.attachCamera(camera);
    testPattern = new Loader();
    testPattern.blendMode = BlendMode.ADD;
    testPattern.alpha = 0;
    holder.addChild(video);
    holder.addChild(testPattern);
    addChild(holder);
    stage.quality = StageQuality.LOW;

    smoothRandomNoise = new BitmapData(300, 10, false, 0);
    smoothRandomNoise.perlinNoise(Math.random()*100, Math.random()*100, 4,
      int((new Date()).date) * int(1000*Math.random()), true, true, 7);
    pointers = new Vector.<Point>(12);
    rands = new Vector.<Number>(12);
    for (var i:int = 0; i < 12; i++)
      pointers[i] = (new Point(
        Math.random()*smoothRandomNoise.width,
        Math.random()*smoothRandomNoise.height));

    var PBJURL:String = "http://actionscriptbible.com/files/badreception.pbj";
    var loader:URLLoader = new URLLoader(new URLRequest(PBJURL));
    loader.dataFormat = URLLoaderDataFormat.BINARY;
    loader.addEventListener(Event.COMPLETE, onBytecodeLoaded);

    var IMGURL:String = "http://actionscriptbible.com/files/testpattern.jpg";
    testPattern.load(new URLRequest(IMGURL));
    testPattern.contentLoaderInfo.addEventListener(Event.COMPLETE, onImgLoad);
  }
  protected function onBytecodeLoaded(event:Event):void {
    shader = new Shader(URLLoader(event.target).data);
    shaderFilter = new ShaderFilter(shader);
    shader.data.noiseImage.input = noise;
    shader.data.srcImage.input = noise;
    shader.data.dimensions.value = [noise.width, noise.height];

    stage.fullScreenSourceRect = noise.rect;
    stage.addEventListener(MouseEvent.CLICK, onFullScreen);
```

```
      stage.addEventListener(Event.ENTER_FRAME, go);
    }
    protected function onImgLoad(event:Event):void {
      testPattern.width = noise.width;
      testPattern.height = noise.height;
    }
    protected function onFullScreen(event:MouseEvent):void {
      if (stage.displayState == StageDisplayState.FULL_SCREEN) {
        stage.displayState = StageDisplayState.NORMAL;
      } else {
        stage.displayState = StageDisplayState.FULL_SCREEN;
      }
    }
    protected function go(event:Event):void {
      noise.noise(getTimer(), 0, 255, 7, true);
      for (var i:int = 0; i < pointers.length; i++) {
        var p:Point = pointers[i];
        p.x = (int(p.x) + 2*(1+i)) % smoothRandomNoise.width;
        rands[i] = smoothRandomNoise.getPixel(p.x, p.y) / 0x00ffffff;
      }

      var sd:ShaderData = shader.data;
      testPattern.alpha = Math.max(0, rands[6] * 3 - 2);
      sd.noisyHDisplace.value = [Math.pow(rands[2],4)*40];
      sd.vRoll.value = [rands[0] * 80 - 20];
      sd.channelSplit.value = [rands[3] * 40 - 10];
      sd.sinHDisplaceAmplitudes.value = [20.0, 2.0, 1.0];
      sd.sinHDisplaceFrequencies.value =
        [rands[2]*2+0.4, rands[10]*4+4, rands[11]*8+8];
      sd.sinHDisplace.value = [Math.pow(rands[2], 8) * 100];
      sd.noiseLayer.value = [Math.pow(rands[10], 4) * 0.8];
      sd.blackoutThresh.value = [rands[2]*0.3];
      holder.filters = [shaderFilter];
    }
  }
}
```

In the example, you use a strip of Perlin noise to produce cycles of natural-looking randomness. You have a set of points you move across the strip of noise at different speeds, sampling the cyclical noise, and storing the random numbers in the rands vector. By picking an entry from this vector with a higher index, you get faster-cycling noise. By raising these values — in the range zero to one — to higher powers with Math.pow(), you make the distribution of the noise cluster toward the edges more. This way, you get more extreme effects less often. Anyway, try it. The effect is akin to Figure 38-1.

FIGURE 38-1

A complex effect given by Example 38-2

Bending Other Data

This section investigates two advanced applications of Pixel Bender: using kernels to crunch different kinds of data efficiently, and asynchronously. These goals are far from mutually exclusive.

As I've said before, even if a Pixel Bender kernel runs on the CPU, it may utilize the CPU better than ActionScript 3.0 would. If you design your programs to break out time-critical vector processing routines into Pixel Bender kernels, you can make better use of the CPU and squeeze out more performance, especially on multicore machines.

Whether or not a kernel runs faster than equivalent ActionScript, the kernel is able to run asynchronously to ActionScript code. For huge data sets, simply freeing up AVM2 to handle the application's interface and input handling is enough of a benefit. Rather than hang while the data is processed, the kernel can execute in one or more threads, firing an event back to ActionScript when its results are ready. In other words, you may not be able to move processing off the CPU, but you can move it off the main thread.

To take advantage of Pixel Bender for generalized data processing, though, your problem should be compatible with Pixel Bender's pipeline. Rather than run your kernel on every pixel of an image, it runs your kernel on every sample of the data. (Actually, the kernel still runs on an image — you just craft this image to store your data.) So the problem should be highly parallelizable. The result for any sample of the data can't rely on the result for another sample. The input has to be one or more sets of data, and the output of the kernel must be a set of data. You must be able to represent the input and output data as sequences or arrays of 32-bit floating-point numbers. Also, you can only use the math functions and data types that Pixel Bender provides, so save it for crunching numbers, vectors, and matrices.

Preparing Data for Pixel Bender

You can use multiple inputs in a kernel, but all inputs must be image types. Recall that image types in Pixel Bender have between one and four channels, which are stored as 32-bit precision floating-point

values between 0.0 and 1.0. Keep in mind that, regardless of the number of channels, an image is still conceptually two dimensional. These attributes of Pixel Bender inputs determine to a great extent the design of your kernel and ActionScript program.

You've already looked at the `ShaderInput` class, which gives your ActionScript code access to the inputs of a Pixel Bender kernel. This interface gives you three ways to provide data, as it accepts three types to its `input` property. The same types are also acceptable types for a `ShaderJob`'s `target` property, where the pixel outputs will be reassembled.

BitmapData

If you're inputting image data in the first place, it's easiest to send in image data as `BitmapData`. If the input uses less than 4 channels, data in these channels will be ignored. Beware that Flash Player's ARGB 8-bit channels will be converted to Pixel Bender's RGBA 32-bit channels, and vice versa. When using `BitmapData` as a `target` type, expect significant precision loss. Additionally, channel values outside the 0.0–1.0 range are clamped when this conversion occurs. One of the benefits of using `BitmapData` is that you don't have to explicitly instruct Flash Player what the sizes of the input or expected output are; dimensions are stored in the `BitmapData` instead.

Vector

Perhaps the best choice for custom, nonimage data, a `Vector.<Number>` requires no conversion. You can both input and output 32-bit floating-point values whether they are in the acceptable range for image data or not. The number of entries in an input or target vector must agree with the input or output of the kernel. A 3×3 image at 4 channels needs 36 `Number`s. If accepting output as a `Vector`, the instance must be instantiated and with enough room to accommodate the expected output. You must instruct the `ShaderInput` what dimensions to interpret a `Vector` as, and you must instruct the `ShaderJob` what dimensions to expect back from the kernel if the target is a `Vector`. These dimensions should not take into account the number of channels. Pixel Bender already knows that.

```
//kernel does image1 -> pixel4

var invec:Vector.<Number> = new <Number>[-4, 2, 100, 0.238];
//out-of-range floats will be respected
var input:ShaderInput = shader.data.inimg;
trace(input.channels); //1
input.input = invec;
input.width = 4; //let's use the input like a long strip
input.height = 1;

var outvec:Vector.<Number> = new Vector.<Number>();
//must be initialized
var job:ShaderJob = new ShaderJob(shader);
job.target = outvec;
job.width = 4;
job.height = 1;
//or try mirroring the input
//job.width = input.width
//job.height = input.height
job.start(true);
trace(outvec.length); //16, 4 by 1 by 4 channels
```

ByteArray

You can also send or receive batches of floating-point data using a `ByteArray`. As with `Vectors`, you won't lose accuracy in the process. Use `writeFloat()` and `readFloat()` and the `ByteArray` cursor to prepare and extract single points of data, keeping in mind that a 32-bit float is 4 bytes. This method is useful when the data you need to process is already tightly packed floats, or when the result of the kernel is to be used in this format, such as a sound sample. You'll again have to instruct Pixel Bender of the intended dimensions when using a `ByteArray` as input or output of a kernel.

```
//kernel does image4 -> pixel4

var input:ShaderInput = shader.data.inimg;
trace(input.channels); //4
input.width = 4;
input.height = 1;
var inbytes:ByteArray = new ByteArray();
for (var i:int = 0; i < input.width * input.height * input.channels; i++) {
  inbytes.writeFloat(i % 4);
}
input.input = inbytes;

var outbytes:ByteArray = new ByteArray();
var job:ShaderJob = new ShaderJob(shader, outbytes, 4, 1);
job.start(true);
trace(outbytes.length / 4); //16 floats (4 is length of a float)
for (outbytes.position = 0; outbytes.position < outbytes.length;) {
  trace(outbytes.readFloat());
  //0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3 4 by 1 by 4 channels
}
```

Accessing Data in the Kernel

To get back at the data inside the kernel, use the sample function `sampleNearest()` as you might use an array lookup in a 2D array. If you decided to store different information in the channels of each pixel, you can get this out, and back in, by swizzling.

Remember that you can use multiple image inputs. Simply pass the image you want to use as input to the sample function, and you can mix in data from multiple sources. After the next section, you'll see how to read and execute shaders, in Example 38-3.

Executing and Monitoring ShaderJobs

Use `ShaderJob` to control how a kernel is run. With `ShaderJob`, it's up to you to provide all the proper inputs and an appropriate target for the kernel's output. These may be in any combination of the types discussed in "Preparing data for Pixel Bender."

To create a `ShaderJob`, initialize it with a `Shader` parameter. You can set the `target`, `width`, and `height` in the constructor or afterward using those parameter names. Before running the job,

you need to specify a valid target for the output, but `width` and `height` may be omitted if this is a `BitmapData` instance.

```
var job:ShaderJob = new ShaderJob(shader);
job.target = outByteArray;
job.width = 100;
job.height = 1;

var job:ShaderJob = new ShaderJob(shader, outBitmapData);
```

A `ShaderJob` may be used in synchronous or asynchronous mode. When in synchronous mode, the job finishes before the next line of ActionScript is run. When using asynchronous mode, a `ShaderEvent.COMPLETE` event is broadcast by the `ShaderJob` as soon as the kernel finishes processing. To start a job, call `start()`, optionally passing `true` to use synchronous execution. Asynchronous mode is the default.

```
job.addEventListener(ShaderEvent.COMPLETE, onComplete);
job.start();

job.start(true);
trace("already finished!");
```

Don't forget to remove your event listeners when an asynchronous job completes.

You can check up on a job's progress with its `progress` accessor (ranges between 0 and 1). Cancel a job if necessary with the `cancel()` method. These, of course, apply only to asynchronous mode `ShaderJobs`.

The Double-Shader Experiment

In Example 38-3, you'll simulate asteroids in a binary star system with not one but two shaders. More accurately, you'll generate a static gravitational force field from two huge gravity wells (the binary star system), then place a set of point masses — in other words, particles — inside this force field and let the simulation run. You'll use a different shader for each of these tasks.

A force field defines the forces acting on each point in the field, and you can easily encode one in an image, by using two of the color channels to encode the x and y components of the force vector at that pixel. Since Pixel Bender is so adept at image manipulation, this is the ideal way to pass off the force information. And the benefit of precalculating this force field is huge. When you're calculating the forces on a given object in the scene — an asteroid, for example — you never actually have to calculate those forces. The forces are entirely dependent on its position in the scene, and finding the force to apply is a simple matter of looking up the object's position in the force field.

Back to the force map itself. With the stars' positions and masses as parameters, Pixel Bender is the perfect way to crunch out a huge force map in no time. For each output pixel, you calculate the cumulative effects of gravity from the two stars.

829

```
<languageVersion : 1.0;>
kernel GravitySimGenerateMap
< namespace : "com.actionscriptbible";
  vendor : "ActionScript 3.0 Bible";
  version : 1;
  description : "Generates a force map from two stars";
>
{
  parameter float3 star0;
  parameter float3 star1;
  parameter float power <defaultValue: float(2.0);>;
  input image4 src;
  output pixel4 dst;

  void evaluatePixel() {
    float2 point = outCoord();
    float2 totalForce = float2(0.0);
    float2 force;
    float dist;

    //unrolled loop iteration 0
    force = star0.xy - point;
    dist = length(force);
    force = normalize(force);
    force *= star0.z / (pow(dist, power));
    totalForce += force;
    //unrolled loop iteration 1
    force = star1.xy - point;
    dist = length(force);
    force = normalize(force);
    force *= star1.z / (pow(dist, power));
    totalForce += force;

    totalForce = clamp(totalForce, -100.0, 100.0) + 100.0;
    totalForce /= 200.0;
    dst = pixel4(0.0, 0.0, 0.0, 1.0);
    dst.rg = totalForce.xy;
  }
}
```

Because Pixel Bender lacks looping structures, use an *unrolled* loop: write out the code for each iteration of the loop, even if it means duplicating lines of code. To keep things relatively concise, you'll stop at two stars. You could easily add more by adding more parameters and more copies of the unrolled loop.

I've parameterized the fall-off power of gravity here in order to exaggerate its effects. Of course, gravity decreases in power proportional to the distance squared. But in Example 38-3, I'll pass in a slightly lower value so that the gravity wells are much more gentle than they otherwise would be.

Because the force map needs to go into an image, I've shifted forces up to all be positive, and in the range 0 to 1.0. Recall that Pixel Bender stores channel values as 0 to 1.0 rather than 0 to 255. The first step of Example 38-3 loads in this kernel, sets its parameters, and executes it with a synchronous ShaderJob. Then it loads in the second shader.

The second shader is even more data-centric. You can use Pixel Bender's parallelism to process large sets of independent data, so in this case you use it to calculate the new position and velocity of a huge set of independent particles. In this shader, you pass in data as a `Vector.<Number>` as discussed in the section "Preparing data for Pixel Bender." Each sample encodes a single point mass. You use four channels to store the particle's x and y position, and the x and y components of its velocity vector. Then you feed this `Vector` into the shader as a $n \times 1$ image, where n is the number of particles.

The shader's code is trivial, if you know the basic equations of motion:

$$d\mathbf{v} = \mathbf{a}\, dt$$
$$d\mathbf{p} = \mathbf{v}\, dt$$

You simply unpack the particle's position and velocity, and update them. You find the force to apply by sampling the force field at the particle's position. To simplify matters, you assume a unit mass and a unit time step.

```
<languageVersion : 1.0;>
kernel GravitySimStep
< namespace : "com.actionscriptbible";
  vendor : "ActionScript 3.0 Bible";
  version : 1;
  description : "Gravity simulation step";
>
{
  parameter float2 scale <defaultValue: float2(1.0, 1.0);>;
  input image3 forcemap;
  input image4 inpoints;
  output pixel4 outpoint;

  void evaluatePixel() {
    //each input pixel represents a point mass (particle) with mass=1
    //inpoint.XY = <x, y> position
    //inpoint.ZW = <x, y> velocity
    float4 particle = sampleNearest(inpoints, outCoord());
    //the forcemap at a pixel stores the force vector at that point
    //forcemap[x, y].RG = <x, y> force vector (normalized... 0.5 = 0)
    float2 force = sampleNearest(forcemap, particle.xy).rg;
    force = (force - float2(0.5)) * scale;
    float2 velocity = particle.zw + force;
    float2 position = particle.xy + velocity;

    outpoint.xy = position;
    outpoint.zw = velocity;
  }
}
```

The shader runs one step of the simulation, so at the end, you set the output pixel with the particle's updated position and velocity. In the ActionScript 3.0 code, you'll run this shader on a large set of particles, once per frame update.

Finally, put these two shaders together in Example 38-3.

`http://actionscriptbible.com/ch38/ex3`

Using ShaderJob on a Data Set

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.geom.*;
  import flash.net.*;
  public class ch38ex3 extends Sprite {
    protected const URL_BASE:String = "http://actionscriptbible.com/files/";
    protected const INITIAL_PARTICLES:int = 1000;
    protected const WELL_GRAVITY_MAX:Number = 5000;
    protected var W:int, H:int;

    protected var generatorShader:Shader;
    protected var stepShader:Shader;
    protected var forcemap:BitmapData;
    protected var particleData:Vector.<Number>; //4 channels: px,py,vx,vy

    protected var bmp:BitmapData;
    protected var bitmap:Bitmap;
    protected const DIM:ColorTransform = new ColorTransform(.9, .9, .9, .95);
    public function ch38ex3() {
      stage.frameRate = 60;
      loadShader("GravitySimGenerateMap.pbj", onGeneratorShaderReady);
    }
    protected function loadShader(url:String, onComplete:Function):void {
      var loader:URLLoader = new URLLoader(new URLRequest(URL_BASE + url));
      loader.dataFormat = URLLoaderDataFormat.BINARY;
      loader.addEventListener(Event.COMPLETE, onComplete);
    }
    protected function onGeneratorShaderReady(event:Event):void {
      event.target.removeEventListener(Event.COMPLETE, onGeneratorShaderReady);
      generatorShader = new Shader(URLLoader(event.target).data);

      forcemap = new BitmapData(stage.stageWidth, stage.stageHeight, false, 0);
      W = forcemap.width; H = forcemap.height;
      var forceBitmap:Bitmap = new Bitmap(forcemap);
      addChild(forceBitmap);
      bmp = new BitmapData(W, H, false, 0);
      bitmap = new Bitmap(bmp);
      addChild(bitmap);
      bitmap.alpha = .99;

      initializeParticles();
      initializeWells();
      loadShader("GravitySimStep.pbj", onStepReady);
    }
    protected function onStepReady(event:Event):void {
      event.target.removeEventListener(Event.COMPLETE, onGeneratorShaderReady);
      stepShader = new Shader(URLLoader(event.target).data);
```

```
      stage.addEventListener(MouseEvent.CLICK,
        function(event:Event):void{initializeParticles()});
      stage.addEventListener(KeyboardEvent.KEY_UP,
        function(event:Event):void{initializeWells()});
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function initializeParticles():void {
      var r:Function = Math.random;
      particleData = new Vector.<Number>();
      for (var i:int = 0; i < INITIAL_PARTICLES; i++) {
        particleData.push(r()*W, r()*H, (r()-0.5)*4, (r()-0.5)*4);
      }
    }
    protected function initializeWells():void {
      var r:Function = Math.random;
      var data:ShaderData = generatorShader.data;
      data.power.value = [1.4];
      data.star0.value = [r()*W/2 + W/4, r()*H/2 + H/4, r()*WELL_GRAVITY_MAX];
      data.star1.value = [r()*W/2 + W/4, r()*H/2 + H/4, r()*WELL_GRAVITY_MAX];
      var shaderJob:ShaderJob = new ShaderJob(generatorShader, forcemap);
      shaderJob.start(true);
    }
    protected function step():void {
      var data:ShaderData = stepShader.data;
      data.forcemap.input = forcemap;
      data.inpoints.input = particleData;
      data.inpoints.height = 1;
      data.inpoints.width = particleData.length/4;
      var shaderJob:ShaderJob = new ShaderJob(stepShader, particleData,
        particleData.length/4, 1);
      shaderJob.start(true);
    }
    protected function draw():void {
      bmp.colorTransform(bmp.rect, DIM);
      bmp.lock();
      for (var i:int = 0; i < particleData.length; i+=4) {
        bmp.setPixel(particleData[i], particleData[i+1], 0xffffff);
      }
      bmp.unlock();
    }
    protected function onEnterFrame(event:Event):void {
      step();
      draw();
    }
  }
}
```

The two bolded sections in the example show how parameters and data are submitted to two different ShaderJobs. In the `stepShader`, two inputs are used. One, `forcemap`, is a `BitmapData` object, while the other, `particleData`, is a `Vector.<Number>`. Note that when you submit a `Vector`,

you must explicitly tell the `ShaderParameter` what dimensions it should assign to the image. Also notice how the `Vector` is a flat list of `Numbers`, every four defining a single particle; these four values become the four channels of the `image4` input inside the shader.

For an even better application of shaders, you could simulate every particle interacting with every other particle, a feat which would be very complex to code traditionally. Every possible combination of two particles would have to be tested for effects on each other, the complexity increasing exponentially with the number of particles. Instead, you could additively "stamp" the gravity well signature for a particle on top of the force field bitmap to generate the force map used for the next frame. This would make an excellent further exercise.

Summary

- Pixel shaders were created to move short, low-level, optimized calculations onto the graphics card and into the hardware-accelerated graphics pipeline.

- Pixel shaders originally let 3D graphics programmers move beyond built-in lighting and texturing. Modern graphics cards are highly programmable.

- Pixel Bender lets you write 2D pixel shaders that work inside Flash Player on the CPU. GPU support may come in future versions.

- Pixel Bender is available on other Creative Suite products, After Effects, and Photoshop.

- Pixel Bender programs, or kernels, run in parallel for every pixel, taking in zero or more images and outputting a single pixel. The pixels are reassembled to form an output image.

- Flash Player loads kernels as bytecode, and then it further compiles the bytecodes at runtime to run on your hardware, including vector instruction sets and multiple cores of your CPU.

- Pixel Bender kernels can operate asynchronously to ActionScript code.

- The kernel language is inspired by GLSL. Pixel Bender on Flash Player is missing several key features like loops. It's simple.

- Kernels have metadata, members (constants, parameters, inputs, and an output), and a single `evaluatePixel()` function.

- The Pixel Bender kernel language provides scalar, vector, matrix, and image types, as well as dozens of top-level functions, mostly for math. Operators and functions are overloaded.

- ActionScript interfaces with a kernel through the `Shader` object, which is associated with the PBJ bytecode as a `ByteArray`.

- `ShaderData`, found in the `data` property of a `Shader`, contains the kernel's metadata, `ShaderInputs`, and `ShaderParameters`.

- ActionScript may run shaders as filters, strokes, fills, blend modes, or standalone jobs.

- Use instances of `BitmapData`, `Vector.<Number>`, or a `ByteArray` full of floats to send data to or receive data from a kernel, including masquerading nonimage data as an image.

Scripting Animation

Animation is one of the serious strong points of the Flash platform. Flash evolved out of an animation tool, and it continues to be used to animate everything from web comics to popular television shows and advertisements. There has always been a big gap between using the Flash IDE to animate by hand and creating animations in ActionScript 3.0. Motion XML, introduced in Flash Professional CS3, helps to bridge this gap. In addition, the Flex framework sports its own animation framework, and there are a number of freely available third-party animation frameworks for ActionScript 3.0.

FEATURED CLASSES

`fl.motion.*`

Understanding Flash Player and Animation

Intuitively, you know that animation is movement. Specifically, animation is movement achieved by sequencing individual still pictures. Your eyes and your brain conspire to interpret these separate pictures, or *frames*, as one fluid motion, through persistence of vision.

Frame Rate

Flash Player, like all things running on a computer, displays static frames in sequence. It attempts to display these frames as fast as the *frame rate* of the movie, which is measured in frames per second. Motion at 30 frames per second (fps) appears fluid, but presenting even more frames per second can help. In both Flex and Flash you have control over the frame rate of the SWF you publish, and I recommend using at least 30 fps unless you have a compelling reason not to.

Note

If video is playing in your SWF, the video still plays at its source frame rate regardless of the frame rate of the host SWF. However, a subordinate SWF loaded into your own SWF plays in the master SWF's frame rate regardless of its own. ■

For comparison, you probably know that films play at 24 fps, and television in the United States plays at about 30 fps. Computer displays can be driven at a great variety of actual refresh rates (usually around 60 fps), but when designing your application, target it for whatever speed you desire.

Flash Player Operation

Flash Player does not always display new frames at the rate you request. The frame rate you set is a goal that Flash Player tries to achieve. The actual frame rate is limited by many variables.

To render a frame, Flash Player must utilize enough CPU time to render the new frame before it is scheduled to display. This can be limited by the complexity of the animation and ActionScript code that is running, the speed of the CPU and other hardware factors (bus speed, memory available, cache misses, and so on), as well as by the operating system and browser. It's a lamented fact that no SWF file running in a browser runs as fast as it does in the standalone Flash Player application. The plug-in architectures available on the various browsers limit the plug-in Flash Player's access to and allotment of resources. The browser and operating system sometimes take further chunks of performance away from plug-ins like Flash Player for the sake of power consumption and other first-class applications.

Executing ActionScript is just part of what goes into rendering a frame. Flash Player must also put together the image you will see. In most cases, Flash Player renders frames entirely with software, as opposed to 3D games and newer operating systems, which offload much of this work onto your graphics card. Because the images that you see are composed using your CPU, composing any one frame can take up a lot of time in addition to any ActionScript running. For a great description of how Flash Player spends its time rendering a frame, see Sean Christmann's article at `http://bit.ly/elastic-racetrack`.

If you set the frame rate of a SWF file to 60 fps, Flash Player has just 1/60 of a second to draw each frame, or 16.67 milliseconds. If the system takes 25 milliseconds to render one of these frames, it's going to show up as soon as it can, around 8 milliseconds late. Already, with one frame late, Flash Player isn't truly playing at 60 fps. If a few frames come in late, you might see that the player's true frame rate is fluctuating. And if every frame takes 25 milliseconds to render, Flash Player will play at 40 fps, regardless of the fact that you asked for 60 fps. In conclusion, the frame rate is not a guarantee, and it is not fixed. It controls how often Flash Player tries to render new frames, not necessarily how often it does.

Flash Player juggles two kinds of control flow: while it's rendering frames, it's also running Action-Script. It's important to understand that for any given frame to render, all ActionScript for that frame must be completed. If, for example, you ran the following code

```
for (var i:int = 0; i < 500; i++) {
    myBlueShape.x = i;
}
```

the display object `myBlueShape` would not appear to move from the left edge of the stage to 500 pixels to the right. Instead, it would appear suddenly at 500 pixels. Because all available ActionScript code is executed before you see the next frame, you can't use synchronous code to make anything

move. Flash Player does execute all 500 iterations of this loop, and each assignment is made in order (x holds the value 0, then 1, then 2, and so on), but nothing makes its way to the screen until all the code is completed. Because Flash Player works this way, you have to be aware of the order that things occur. Although action B might occur far after action A in code, the user cannot see this gap in time represented on the screen. And if you set a variable to something in reaction to an event, there may be frames that execute for which that variable is not yet set.

A side effect of the order in which Flash Player operates is that involved computation can slow down or halt the rendering of new frames. This is also the reason that, when an uncaught exception aborts all ActionScript on a frame, the results can be fatal. Skipping to ActionScript executed in the next frame can ignore hundreds or thousands of lines of code.

ActionScript that you write isn't concurrent. It can't be threaded or fork off processes. It executes from top to bottom. However, native Flash Player API calls may execute in parallel behind the scenes, like network accesses. You can use some asynchronous events like those broadcast by `Timer` (see Chapter 22, "Timers and Timer-Driven Programming") and the `Event.ENTER_FRAME` event to schedule ActionScript to execute after one or more frames are rendered. Correspondingly, you can use `updateAfterEvent()` methods to request a frame to draw immediately after ActionScript processing finishes, regardless of when the next frame is scheduled to render.

These asynchronous events are the primary way that ActionScript gets executed in frames after the first one; as such, they can make things animate.

Animating with Code

At the core of any programmatic animation is some event that is fired at regular intervals. This enables ActionScript to be executed repeatedly across time, rather than all at once. If you can move an object a tiny bit at a time at regular intervals, you have achieved animation. I will show simple examples of building animation from scratch using only `Event.ENTER_FRAME` or a `Timer`. Many examples in previous chapters use these simple time-based animations, but in this section you'll learn how to refine those methods.

Creating these examples yourself will give you an understanding of what's happening behind the scenes and enable you to create your own animation code when premade animation packages do not suffice.

This chapter will only get you started with animation principles with ActionScript. For a guide to creating algorithmic motion in code, including basic physics, particles, and forward and inverse kinematics, take a look at *Foundation ActionScript 3.0 Animation: Making Things Move!* by Keith Peters (Friends of ED, 2007).

Animating by Time

Animating using a `Timer` object gives you some distinct benefits. When you are using a `Timer` object, you can set the frequency at which new frames are drawn completely independently of the published frame rate of the SWF. You can employ this to change the frame rate of programmed animations without interfering with timeline animations created by the Flash authoring tool or running two animations at different frame rates. For example, you can animate the main characters of a game at 30 fps but save some CPU time by animating the background at 15 fps.

It's possible for events from a `Timer` object to fire more frequently than the published frame rate of a movie, and you can ensure that the results of these events are made visible by calling

`updateAfterEvent()` on the event object. This can actually make Flash Player play faster than the published frame rate. However, updating more frequently than the frame rate without showing the results visually is a waste.

Example 39-1 animates a `DisplayObject` moving the object to the right at whatever frame rate is assigned to the `fps` variable (here, it is 60), regardless of the published frame rate (set to 1 here).

EXAMPLE 39-1 `http://actionscriptbible.com/ch39/ex1`

Timer-Driven Animation

```
package {
  import flash.display.*;
  import flash.events.TimerEvent;
  import flash.utils.Timer;
  [SWF(frameRate="1")]
  public class ch39ex1 extends Sprite {
    protected var ball:DisplayObject;
    public function ch39ex1() {
      ball = new Ball();
      ball.y = stage.stageHeight/2;
      addChild(ball);

      // fps = frames/second
      var fps:int = 60;
      // 1/fps = seconds/frame, 1/fps*1000 = milliseconds/frame
      var mspf:int = Math.round(1 / fps * 1000);

      var t:Timer = new Timer(mspf);
      t.addEventListener(TimerEvent.TIMER, onTimer);
      t.start();
    }
    protected function onTimer(event:TimerEvent):void {
      ball.x += 0.5;
      event.updateAfterEvent();
    }
  }
}
import flash.display.Shape;
class Ball extends Shape {
  public function Ball(color:uint = 0xff0000, size:Number = 50) {
    graphics.beginFill(color);
    graphics.drawCircle(0, 0, size);
  }
}
```

The `onTimer()` method is called every 17 milliseconds, or about 60 times a second, or as close to that as Flash Player can manage. After the method modifies the `x` property of the display object, it asks Flash Player to redraw the screen. You'll see that even a SWF published at 1 fps runs the animation at the requested 60 fps.

Animating by Frames

You can tie the frame rate of programmatic animation to the frame rate of the movie by using the `Event.ENTER_FRAME` event broadcast by display objects when Flash Player advances to the next frame. This is a less invasive way of scheduling animations: it lets the host SWF determine the frame rate, and it doesn't force frames to be drawn when they wouldn't normally be drawn.

Note

You can set, and even change, the frame rate at runtime by assigning a `Number` (in FPS) to `stage.frameRate`. ∎

Example 39-2 animates a display object toward the right just like Example 39-1, but using the SWF's own frame rate.

EXAMPLE 39-2 `http://actionscriptbible.com/ch39/ex2`

Frame-Driven Animation

```
package {
  import flash.display.*;
  import flash.events.Event;
  [SWF(frameRate="60")]
  public class ch39ex2 extends Sprite {
    protected var ball:DisplayObject;
    public function ch39ex2() {
      ball = new Ball();
      ball.y = stage.stageHeight/2;
      addChild(ball);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      ball.x += 0.5;
    }
  }
}
import flash.display.Shape;
class Ball extends Shape {
  public function Ball(color:uint = 0xff0000, size:Number = 50) {
    graphics.beginFill(color);
    graphics.drawCircle(0, 0, size);
  }
}
```

Animation and Speed

Both of the previous examples move the display object at a particular frequency: 1 pixel every 2 frames. But in both approaches, the speed of the motion depends on the frame rate. To determine how fast the object appears to be moving, you can factor frames out of the equation:

$$\frac{\text{pixels}}{\text{second}} = \frac{\text{pixels}}{\text{frame}} * \frac{\text{frames}}{\text{second}}$$

839

Applying this equality, the object moves at 30 pixels per second at 60 fps, and 15 pixels per second at 30 fps. Again, the number of frames per second determines how fluid motion is; pixels per second is an absolute measure of speed.

The formula tells why it can be dangerous to program animation in terms of frames. In the timer example, changing the SWF's frame rate doesn't actually change the frame rate of the programmatic animation, and the object should appear to move at the same speed. In the enterFrame example, changing the SWF's frame rate directly changes the speed of the animation. In both cases, when Flash Player can't keep up with the requested frame rate, the speed of the animation is affected.

A better approach is to construct an update function that moves the target at the desired speed by linking the amount of motion to the amount of time that has actually elapsed since the last frame was rendered. If you ignore the frame rate of the SWF and the frequency of the Timer interval and measure the time yourself, you can adjust for inadequate or changing frame rates. Although you can never guarantee that the frame rate will keep up, you can guarantee that if it doesn't, the speed of the motion will remain the same.

You can use the getTimer() function from the flash.utils package to measure real elapsed time. The function returns the number of milliseconds since the SWF started playing. Remember that:

$$v = \frac{\Delta p}{\Delta t}$$

Or, velocity equals the change in position over the change in time. To animate at a constant velocity, you must determine how far to move the object at every frame. To find that out, simply rearrange the equation:

$$\Delta p = v\ \Delta t$$

Every frame, you simply find out how much time has elapsed (Δt), multiply by the velocity (v), and move the object by that distance (Δp).

Example 39-3 keeps track of the change in time to move the object a constant speed regardless of frame rate.

EXAMPLE 39-3   http://actionscriptbible.com/ch39/ex3

Framerate-Independent Motion

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.text.*;
  import flash.utils.getTimer;
  public class ch39ex3 extends Sprite {
    protected const SPEED:Number = 50 / 1000;
    //50 pixels per second * 1sec / 1000ms = 50/1000 pixels per millisecond
    protected var ball:DisplayObject;
    protected var tf:TextField;
    protected var lastTime:int;
    public function ch39ex3() {
      ball = new Ball();
      ball.y = stage.stageHeight/2;
      addChild(ball);
      tf = new TextField(); tf.width = 0; tf.height = 14; tf.x = tf.y = 5;
      tf.autoSize = TextFieldAutoSize.LEFT;
```

```
        tf.backgroundColor = 0; tf.background = true;
        tf.defaultTextFormat = new TextFormat("_typewriter", 11, 0xffffff);
        addChild(tf);
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
        lastTime = getTimer();
    }
    protected function onEnterFrame(event:Event):void {
        //mess things up by changing the frame rate randomly
        stage.frameRate = Math.random() * 45 + 1;

        var time:int = getTimer();
        var dt:Number = time - lastTime;
        ball.x += SPEED * dt;
        lastTime = time;

        tf.text = stage.frameRate.toFixed() + " fps";
        if (ball.x > stage.stageWidth) ball.x = 0;
    }
  }
}
import flash.display.Shape;
class Ball extends Shape {
  public function Ball(color:uint = 0xff0000, size:Number = 50) {
    graphics.beginFill(color);
    graphics.drawCircle(0, 0, size);
  }
}
```

You should apply this principle, especially in games and simulations, to keep differences in computers' speed from affecting the gameplay or visual feedback. Even when things aren't moving at constant speeds, it's a great idea to move them based on actual time elapsed rather than time derived from a frame rate, which may not be accurate.

Animating Using Flash Professional

The Flash authoring tool enables you to create timeline animations that are compiled into the SWF and play out without ActionScript code. You can find out how to use Flash Professional to create timeline animations in any Flash book such as *Flash CS4 Professional Bible* by Robert Reinhardt and Snow Dowd (Wiley, 2009). Timeline animations are easy to create, but in the context of an ActionScript-driven project, they are inflexible.

By using code to create your animations, you can modify animations on-the-fly, make them more interactive, apply them to different kinds of targets, and even load them externally.

Flash Professional (versions CS3 and later) comes with a new animation framework that can represent animations as chunks of XML, called *motion XML*. With the addition of E4X, this means you can construct and modify complex animations deftly. It also makes animations convenient to store and load. The icing on the cake is a set of Flash commands that bridge animations you create on the timeline

with motion XML. You can create an animation on the timeline, convert it into XML, optionally modify it in ActionScript to meet your needs, play the animation back without a timeline using the Flash motion package, and even go round-trip, converting motion XML back into timeline animations. You can find these commands in the Commands menu of Flash Professional.

Of course, you can also create animations using motion XML you create by hand, using the `Animator` class to consume the XML. However, you should also consider other animation libraries if you're going to be creating animations in code, because most of them can describe animations far more concisely.

Preliminarily, I'll review some animation terminology as it relates to Flash animation. Then I'll introduce the structure of motion XML, and finally I'll investigate how to use the `fl.motion` package.

Review: Tweens, Keyframes, and Easing

Two important terms are used consistently to describe animations. Because this terminology carries over into motion XML and ActionScript, it begs a small review.

The pioneers of animation quickly discovered a concept that carries through to all modern animation processes: you have to draw a *lot* of pictures, and some aren't as important as others. For example, picture Bugs Bunny sneezing. His face goes from normal, to all twisted up, to forcefully expelling air and spittle. You can, in your mind's eye (hey, he's still under copyright), picture these three key states. But if the act of sneezing takes a whole second of film, then in addition to those three key frames, you will have to draw all 21 other pictures that go in between: the frames that move his face at the right speed through those key positions.

The key frames are called *keyframes*, and the in-between frames are called *tweens*. Animation is a ton of work, so early on in animation studios, it became the lead animator's responsibility to draw those key frames, and it was up to the junior animators to fill in the in-between frames. Nowadays, animation is still a ton of work, and in-between frames are often sent to third parties to finish. Now computers have made things easier for everyone. With Flash Professional and other animation packages, you can set up two keyframes and have the system tween between them. No interns necessary!
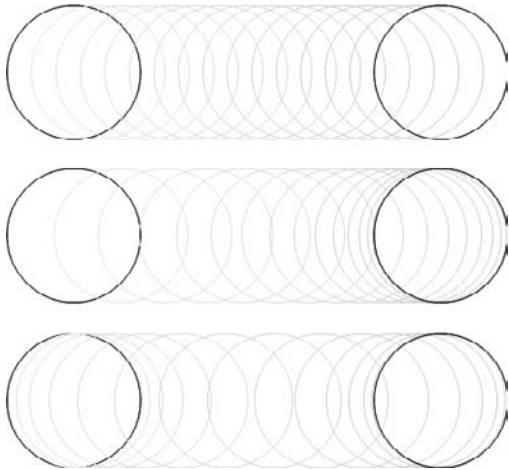
Your brain has adapted to intuitively recognize the effects of Newton's laws. Due to inertia, things always accelerate and decelerate rather than instantly switch speeds. Thus, a linear tween between two positions almost always looks unnatural. Although a linear tween can move an object from point A to point B smoothly, it does so at a constant speed; so at the beginning of the tween, the object instantly changes from no speed to the average speed, and at the end of the tween, the object instantly stops. Thankfully, you can ease into and ease out of tweens, accelerating at the beginning and decelerating at the end, to make the motion look more natural, as shown in Figure 39-1. The change in speed through a tween is called its *easing*, and it can be set by the Flash authoring tool as well as animations created in ActionScript.

Introducing Motion XML

Motion XML is composed of a few simple elements in a predictable structure. Although animating in the timeline can provide for some awkwardly precise decimal values, motion XML should be on the whole easy to read and modify.

Figure 39-1: Three animations: one with no tweening, one with easing out at the end, and one with easing in and out

Each capitalized tag found in motion XML corresponds to a class, usually one from the `fl.motion` package. In the root tag, `<Motion>`, XML namespaces are imported that help you identify where the classes come from. Lowercase tags indicate that the child of the tag is assigned to a property on the parent object. Attributes of a tag also become properties of the object that tag will create. For example,

```
<Sandwich layers="4">
    <toppings>
        <Lettuce/>
        <VeggieMeat/>
    </toppings>
</Sandwich>
```

creates a `Lettuce` object and a `VeggieMeat` object inside the `toppings` property of a `Sandwich` object, like this:

```
var sandwich:Sandwich = new Sandwich();
sandwich.layers = 4;
sandwich.toppings = [new Lettuce(), new VeggieMeat()];
```

This naming convention should be familiar to Flex users. I'll call nodes that correspond to objects *object nodes*, and nodes that correspond to properties *property nodes*. This structure should help you understand how motion XML is converted into objects in the `fl.motion` package, but you don't have to understand it to create and use motion XML. In fact, you can simply use the Flash authoring environment to generate motion XML for you. The package adapts to many styles of use.

The basic structure of motion XML is as follows. Namespaces and attributes are omitted for clarity, and all optional nodes are shown.

```
<Motion>
    <source>
        <Source>
            <dimensions>
                <Rectangle/>
            </dimensions>
            <transformationPoint>
                <Point/>
            </transformationPoint>
        </Source>
    </source>
    <Keyframe>
        <color>
            <Color/>
        </color>
        <tweens>
            <ITween/>
        </tweens>
        <filters>
            <BitmapFilter/>
        </filters>
    </Keyframe>
</Motion>
```

Before I go into the details of each node, here's an overview. All motion XML is contained inside a root `Motion` node. The `Motion` node optionally contains a `source` property node containing a `Source` object node. Following the `source` property node is the bulk of the motion XML, a series of any number of `Keyframe` object nodes.

Each `Keyframe` can set any combination of transformation properties, color transformations, and filters. The color transformations are encoded in the `color` property node and the `Color` object node, and the filter settings for a given keyframe are encoded as object nodes inside a `filters` property node. The node name of a filter object node is equivalent to the name of the filter class.

A `Keyframe` also defines the easing applied to the keyframe by including tween object nodes in a `tweens` parameter node. All tweens implement `ITween`. You can theoretically include more than one tween object node to tween different properties separately (a situation you can't replicate using timeline tweens).

The Motion Object

The root node of any motion XML, the `<Motion>` node sets up the XML's namespaces, associating each namespace with a package so that child nodes scoped to a namespace can be mapped to valid classes within the package. Additionally, the `Motion` object defines the duration of the animation in frames.

```
<Motion duration="14" xmlns="fl.motion.*" xmlns:geom="flash.geom.*"
 xmlns:filters="flash.filters.*"></Motion>
```

This sample node defines an animation that will last 14 frames. It also scopes itself and all child nodes with no explicit namespace to the `"fl.motion.*"` namespace. This helps the animation framework know that when you write a `<Keyframe>` node, it describes an instance of the class `Keyframe`

in the `fl.motion` package. For more information about XML namespaces, see Chapter 11, "XML and E4X."

Running motion XML you create is covered in the section "Using the Flash motion package." To follow along with the examples, create an object to animate, and use a new `Animator` object to execute the motion XML you create. Remember that thanks to E4X, you can type XML directly into your code:

```
var ball:Sprite = new Sprite();
ball.graphics.beginFill(0xff0000);
ball.graphics.drawCircle(0, 0, 20);
ball.graphics.endFill();
addChild(ball);

var motionXML:XML = <Motion duration="14" xmlns="fl.motion.*"
  xmlns:geom="flash.geom.*" xmlns:filters="flash.filters.*"></Motion>;

var animator:Animator = new Animator(motionXML, ball);
animator.play();
```

This particular example doesn't do anything but set up the motion XML, so it won't animate the ball. But you can apply this example snippet to the following motion XML examples to see them in action.

The Source Object

Stored in the `source` property of the `Motion` object, the `Source` object is a comprehensive record of the context in which motion XML was created. It can help Flash Professional map imported XML back onto display object instances. For custom motion XML, you can omit it with no ill effect.

Properties that a `Source` object stores about its contents include the display object's type, instance name, and symbol in the library; the frame rate of the containing document; and the initial position, scale, rotation, and skew of the display object.

The most useful property of a `Source` object, and the main reason to use a `Source` node in custom motion XML, is the `transformationPoint` property. This property stores a `Point` object, which represents the location around which transformations in the animation happen; it is equivalent to the position of the transformation point (the white circle with a black stroke) in Flash Professional. Setting this can drastically change the outcome of rotation, scaling, and skewing.

The `Point` object in the `transformationPoint` property is normalized, or scaled to a unit square. Regardless of the size of the object you are animating, (0, 0) corresponds to the upper-left corner, and (1, 1) corresponds to the lower-right corner.

The following is an example motion XML with only a `source` property:

```
<Motion duration="5" xmlns="fl.motion.*"
        xmlns:geom="flash.geom.*" xmlns:filters="flash.filters.*">
    <source>
        <Source frameRate="12" x="145" y="81" scaleX="1" scaleY="1"
         rotation="0" elementType="movie clip" symbolName="Square">
            <dimensions>
                <geom:Rectangle left="0" top="0" width="44" height="44"/>
            </dimensions>
```

```
                        <transformationPoint>
                            <geom:Point x="0.5" y="0.5"/>
                        </transformationPoint>
                    </Source>
                </source>
            </Motion>
```

Notice that the `<geom:Point>` and `<geom:Rectangle>` nodes are in the `geom` namespace, which refers to `"flash.geom.*"`. This lets the animation framework know to convert the nodes into a `flash.geom.Point` and a `flash.geom.Rectangle`.

Again, the `source` attribute node is optional. You can leave it out, or you can include a `Source` node with only a `transformationPoint` attribute node.

The Keyframe Object

The `<Keyframe>` node corresponds to the `fl.motion.Keyframe` class. These objects represent a single keyframe. At any keyframe, you can set a slew of properties; the change in these properties is tweened when the animation plays.

A simple `Keyframe` might be represented in motion XML as follows:

```
<Keyframe index="13" x="100" y="78" scaleY="1" rotation="70"/>
```

Attributes of the `<Keyframe>` node, shown in Table 39-1, determine the time at which the keyframe is set, the basic transformations of a display object at that point in time, and various special behaviors.

Additionally, several attributes provide finer control for graphics symbols, motion paths, and other flags that are particular to the Flash authoring environment.

The `Keyframe` object can also contain a `color` property, a `filters` array, and a `tweens` array. These are represented in motion XML as child nodes.

TABLE 39-1

Attributes of the <Keyframe> Node

Attribute Name(s)	Type	Usage
`index`	`int`	The frame at which this keyframe is set.
`label`	`String`	Sets an accessible frame label at this keyframe.
`x, y`	`Number`	Position of the target at this keyframe.
`scaleX,scaleY`	`Number`	The horizontal and vertical scale of the target at this keyframe.
`tweenScale`	`Boolean`	When set to `false`, ignores scaling.
`skewX,skewY`	`Number`	The horizontal and vertical skew of the target at this keyframe.

Attribute Name(s)	Type	Usage
`rotation`	`Number`	The rotation, in degrees, of the target at this keyframe.
`rotateTimes`	`uint`	Sets rotation by number of revolutions.
`rotateDirection`	`String`	Determines direction of rotation (''auto'' by default). Defined by constants in `fl.motion.RotateDirection`.
`orientToPath`	`Boolean`	Whether the target rotates to match its path.
`blank`	`Boolean`	If this is an empty keyframe.
`blendMode`	`String`	Changes the blend mode of the target at this keyframe.

The Color Object

The `fl.motion.Color` class, stored in the `color` attribute of a `Keyframe`, is used to tween colors and transparency. A `Color` object encodes a color transform at a point in time in much the same way that a `flash.geom.ColorTransform` object does. In fact, `Color` extends `ColorTransform`, adding the ability to easily construct color transformations that match the brightness and tint controls in the Flash authoring environment.

To set a control point for any of the properties of a color transform, set the corresponding attribute on the `<Color>` node. For example, this motion XML turns an object's brightness all the way down and then fades it out.

```
<Motion duration="90" xmlns="fl.motion.*" xmlns:geom="flash.geom.*"
xmlns:filters="flash.filters.*">
    <Keyframe index="0">
        <tweens>
            <SimpleEase/>
        </tweens>
    </Keyframe>
    <Keyframe index="45">
        <color>
            <Color brightness="-1"/>
        </color>
        <tweens>
            <SimpleEase/>
        </tweens>
    </Keyframe>
    <Keyframe index="90">
        <color>
            <Color alphaMultiplier="0"/>
        </color>
    </Keyframe>
</Motion>
```

The following attributes are supported from `ColorTransform`:

- `redMultiplier`
- `greenMultiplier`
- `blueMultiplier`
- `alphaMultiplier`
- `redOffset`
- `greenOffset`
- `blueOffset`
- `alphaOffset`

For more detail on those properties, see Chapter 34, "Geometric and Color Transformations." The `Color` class adds the following properties to the preceding list:

- `brightness:Number` — A percentage of brightness between −1 and 1.
- `tintColor:uint` — A color to tint to.
- `tintMultiplier:Number` — The amount of tint to apply, between 0 and 1.

These properties control the color of the target, like the Brightness and Tint controls in the Color area of the Properties panel in Flash Professional.

Filter Objects

Filters can be set on any keyframe by adding the appropriate filter objects to the `filters` property of the `Keyframe` object. In motion XML, this consists of creating a tag in the `filters` namespace with the name of the filter you want and the properties of the filter encoded in attributes. If you have the same kind of filter in two or more adjacent keyframes, you can animate properties of the filter. However, you must always include *all* properties of the filter: default values for omitted properties are picked up from the defaults of the filter class rather than the last keyframe in which they were set.

```
<Keyframe index="4">
    <tweens>
        <SimpleEase ease="0"/>
    </tweens>
    <filters>
        <filters:GlowFilter blurX="12" blurY="12"
          color="0xFFFFFF" alpha="1" strength="1" quality="2"
          inner="false" knockout="false"/>
    </filters>
</Keyframe>
```

The ITween Objects

For the properties you set on a `Keyframe` object to be automatically interpolated, the `Keyframe` must contain at least one `ITween` object that defines how to tween the values into the next keyframe. These tweens come in four types: `SimpleEase`, `CustomEase`, `FunctionEase`, and `BezierEase`.

A `SimpleEase` tween can be linear, an ease in, or an ease out, depending on the value assigned to its `ease` property. When omitted, this property defaults to 0, a linear tween with no easing. This kind of tween is the same as the one created by the Ease slider in the Flash authoring environment.

- `<SimpleEase/>` — linear easing; constant speed throughout the tween.
- `<SimpleEase ease="-1"/>` — 100% ease in; quadratic acceleration in the beginning of the tween.
- `<SimpleEase ease="1"/>` — 100% ease out; quadratic deceleration at the end of the tween.

A `CustomEase` tween follows a Bezier curve to interpolate values between the two keyframes. You can declare any curve you want starting at (0, 0) and ending at (1, 1). The x-axis represents time, where 0 is exactly at the keyframe whose ease you are declaring, and 1 is at the time of the following keyframe. The y-axis represents the progress of the tween, where 0 is all the values as set at the first keyframe, and 1 is the end values as they are set in the next keyframe. If you draw a straight line from (0, 0) to (1, 1), that's a linear ease. The steep vertical parts of your curve animate very fast (lots of value change over little time change), and flat parts of your curve slow down or freeze the animation (very little change over lots of time).

You can draw these curves using the Custom Ease In / Ease Out panel in the Flash authoring environment, and you can export them to motion XML. Because this enables you to visually draw and preview the curve, it's much easier than writing `CustomEase`s manually. Following is a simple example of a keyframe that eases in at the beginning of a tween and eases out at the end of it:

```
<Keyframe index="4">
    <tweens>
        <CustomEase>
            <geom:Point x="0.5" y="0"/>
            <geom:Point x="0.5" y="1"/>
        </CustomEase>
    </tweens>
</Keyframe>
```

Note that the control points are defined with `Point` object nodes. Also, the first (0, 0) and last (1, 1) points are implied. The two points are not control points themselves, but the right handle of the control point at (0, 0), and the left handle of the control point at (1, 1). The curve defined by this `CustomEase` is shown in Figure 39-2.

Other than the first and last point, each control point in a `CustomEase` curve is defined by 3 `Point` objects: the location of the incoming handle, the location of the point, and the location of the outgoing handle. So a more complex curve with 3 control points in the middle would be defined by 11 `Point` objects: 3 for each of the control points, 1 for the outgoing handle of (0, 0), and 1 for the incoming handle of (1, 1).

A `FunctionEase` tween uses a formula to determine the easing of the tween. All the well-known easing functions by Robert Penner are included in the animation framework in the `fl.motion.easing` package. Simply assign the fully qualified name of a function with the correct signature to this object's `functionName` property to apply it:

```
<FunctionEase functionName="fl.motion.easing.Bounce.easeIn"/>
```
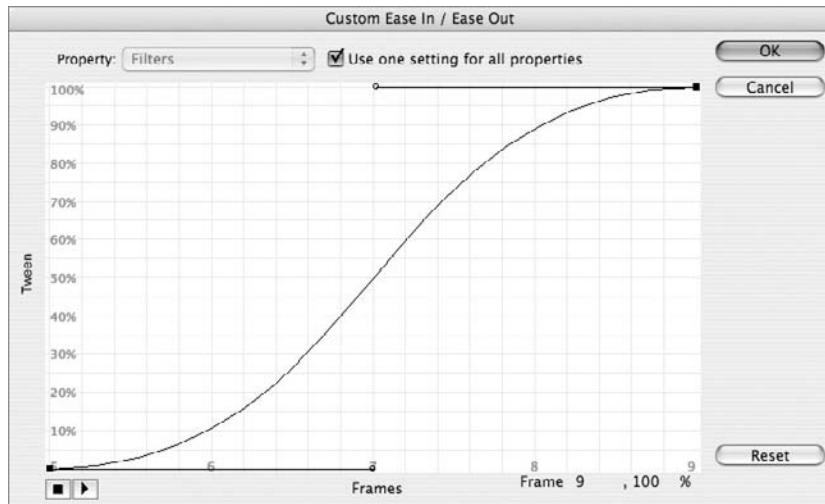
Note
You can read more about Robert Penner's easing equations on his site at `www.robertpenner.com/easing/`. ∎

Figure 39-2: Curve of a CustomEase that eases in and out

For the motion XML to be able to find the function, you must also remember to import the easing class and ensure it's compiled in. You can do this simply by referencing it:

```
import fl.motion.easing.Bounce;
Bounce; //force the Bounce class to compile into the SWF
```

A `BezierEase` tween is similar to a `CustomEase` tween. They both depend on a user-supplied Bezier curve to fully define the value at every point of time. The `BezierEase` tween, however, uses unscaled values of y: the y-axis represents the literal value that the tweened property will take on, rather than the interpolation between the values set out in the keyframes. Therefore, you can use this kind of tween to blow the value out of the range that would otherwise be defined by the endpoints.

Like the `CustomEase`, however, the first and last points are still implied, and the x-axis is still scaled to unit time between the current keyframe (0) and the next (1).

Using the Flash Motion Package

To use the motion XML that you write or copy from the timeline, you need to use the Flash motion package. You can use the classes in this package by themselves or describe animations entirely in motion XML and let the motion package automatically instantiate all the objects you described in XML. For this approach, the main entry point to the motion package is the `fl.motion.Animator` class.

Caution

The Flash motion package, when compiled into your SWF, adds a little more than 12KB to the size of the SWF. This is fairly heavy for an animation toolkit, so if size is a great concern, you might want to consider alternatives. ■

Version

FP9. The Flash motion package requires Flash Player version 9.0.28.0 or later. ■

To create a new animation from motion XML, pass the motion XML and the display object you wish to animate to the `Animator` constructor:

```
var anim:Animator = new Animator(myMotionXML, myDisplayObject);
```

When you have an `Animator`, you're set. You can use some intuitively named methods to control the playback of your new animation, detailed in Table 39-2.

TABLE 39-2

Methods of the Animator Class

Method	Function
`play():void`	Starts playing the animation.
`stop():void`	Stops playing the animation and returns to the first frame.
`end():void`	Stops playing the animation and moves to the last frame.
`pause():void`	Pauses the animation until `resume()` is called.
`resume():void`	Resumes a paused animation.
`nextFrame():void`	Proceeds to the next frame in the animation.
`rewind():void`	Jumps to the first frame in the animation.

Additionally, there is a static method, `Animator.fromXMLString()`, that creates new `Animator` objects without using the constructor. It accepts a `String` containing XML rather than an `XML` object.

You can set the transformation point of the target display object by assigning it to the `transformationPoint` property of the `Animator` object, allowing you to avoid using the `Source` object entirely.

The `time` property of an `Animator` object not only lets you check the current progress of an animation in frames, but it lets you skip to a specific frame by writing to the property.

You can use the `autoRewind` property and the `repeatCount` to set up a looping animation.

The `Animator` object also broadcasts some useful events, described in Table 39-3. All these events are of the type `fl.motion.MotionEvent`.

Any animation structure that you could build in motion XML, you can build by creating and composing classes, but doing so is painfully verbose. Instead, use E4X to set up those relationships in XML. This isn't to say that there is no reason to use the classes in the `fl.motion` package by themselves. You can exploit some of the work that the motion package has to do internally, to save yourself some time. For instance, the `Color` class has a method, `interpolateColor()`, that can blend any two colors.

851

TABLE 39-3

MotionEvent Events

Constant	Purpose
`MotionEvent.MOTION_START`	The motion has started playing.
`MotionEvent.MOTION_END`	The motion has ended, of its own accord or by a call to `stop()` or `end()`.
`MotionEvent.TIME_CHANGE`	The animation is about to work on the next frame. The `time` property has been updated.
`MotionEvent.MOTION_UPDATE`	The animation finished working on a frame and updated the screen.

The real gem is `MatrixTransformer`, a class that does some of the heavy lifting for the motion package. You can use this class to set skews, transform objects around points other than their registration point, and work in radians, all with ease. This is a nice bonus, even if your project doesn't use the rest of the `fl.motion` package to animate.

Animating Using Flex

The Flex framework has an effects system that works brilliantly with styles and states. It enables you to create one-off effects or reusable ones, in ActionScript 3.0 or MXML. Unlike the Flash motion package, the effects package provides support for sequential and parallel animations. It also comes with a few base effect types that you can easily customize. Creating entirely custom effects using the Flex 2 effects framework is less intuitive, however.

If you are using Flex, I recommend you tie into the `mx.effects` package. You can find more information about effects in Flex in the Flex documentation or a book such as *Programming Flex 3*, by Chafic Kazoun and Joey Lott (O'Reilly, 2008) or *Flex 3 Cookbook*, by Joshua Noble and Todd Anderson (O'Reilly, 2008).

Choosing a Third-Party Animation Toolkit

In addition to the choices enumerated above, there are always a growing number of excellent third-party animation packages. Which one you use is usually a matter of personal choice; get to know one inside and out, and it's an obvious choice for any new project.

Your choice of animation toolkit will likely depend on only a few factors:

- syntax — The best advertisement for any code is that you enjoy using it.
- features — Features like animating filters, color transforms, good event management, and event sequencing can all be important.
- speed — You probably want the library to run animations as fast as possible, leaving the rest of your code lots of CPU to spare.
- size — This may only be a real concern for you if you're programming a banner with a strict size limit, but the size of a library's compiled code can be an issue.

Which animation library to use is a constant source of debate (mostly civil) and is as old as time. So there are already some interesting comparisons out there. This article by Mims Wright, author of the KitchenSync library, compares some of the major animation packages, including pros and cons, and syntax samples: `http://bit.ly/mims-tweening-lib-review`. And there are at least two speed comparisons available online: one by Moses Gunesch, author of Go at `http://go.mosessupposes.com/?p=5`; and one by Jack Doyle, author of TweenLite at `http://blog.greensock.com/tweening-speed-test/`.

Table 39-4 is a nonexhaustive list of free animation libraries available in ActionScript 3.0. Maybe you can tell, but I use TweenLite. New libraries are in development all the time, and new features are being added to most of these engines.

TABLE 39-4

Free ActionScript 3.0 Tweening Libraries

Name	Homepage	Comments
TweenLite	`http://blog.greensock.com/tweenlite/`	Fast, light, modular and featureful
Tweener	`http://tweener.googlecode.com/`	Excellent overall engine
KitchenSync	`http://kitchensynclib.googlecode.com/`	Excellent OO design
Eaze	`http://eaze-tween.googlecode.com/`	Nice chaining syntax
gTween	`http://gskinner.com/libraries/gtween/`	
AS3 Animation System	`http://boostworthy.com/blog/?p=170`	
Grape	`http://grape-as3.googlecode.com/`	Specifically for animating complex paths
Twease	`http://twease.googlecode.com/`	
Tweensy	`http://tweensy.googlecode.com/`	
BetweenAS3	`http://www.libspark.org/wiki/BetweenAS3/en`	
Go	`http://goasap.org/`	An extensible framework for developing animation tools

Summary

- Animation changes properties over time.
- Flash Player is running frames as well as ActionScript.
- Flash Player has to finish all the ActionScript on a frame before it can draw the next one.

- The frame rate of a SWF is not a guarantee.
- You can use asynchronous code to create motion.
- Two approaches are `enterFrame` events and `Timer`.
- `Timer` animation is more flexible, but it can be wasteful if you run faster than the frame rate and don't force screen updates.
- Use the actual time differential as reported by `getTimer()` to keep speed constant regardless of frame rate.
- Keyframes are the important frames, and tweens automatically fill the in-between frames.
- Easing can modulate the speed of tweens.
- Flash has a motion package that has feature parity with the timeline.
- The motion package lets you use motion XML, which can be exported from the timeline and imported to the timeline, as well as modified and written from scratch with E4X.
- Motion XML is translated into objects and properties with the same names as the nodes and attributes.
- `Motion` contains an optional `Source` and `Keyframes`.
- `Keyframe` objects can animate anything the timeline can: transformations, color, and filters.
- The motion package supports four ways to describe easing.
- Wrap motion XML in an `Animator` to use it.
- The Flex framework has a totally different approach to animation than the Flash motion package.
- Third-party animation packages are available that might be better for the job, depending on your goals.

Advanced 3D

I n Chapter 15, "Working in Three Dimensions," you learned how to place display objects in 3D space, using what I'll call the 3D display list API. There I introduced perspective projection, the 3D coordinate system, and points and vectors in three dimensions. If all went well, this was a gentle introduction to 3D concepts, and the 3D display list API was a simple and natural interface for 3D work.

Then, in Chapter 34, "Geometric and Color Transformations," you learned about 3D transformation matrices. You delved much deeper into the mathematical principles that enable 3D. You learned how matrix multiplication concatenates transformations and matrix-vector multiplication projects vectors (when the vectors are written as row and column matrices). You didn't use this for any additional rendering techniques, but you did see how to manipulate 3D data such as points and transformations.

In this chapter, you'll put together your grasp of 3D gained in Chapter 15 with your theoretical understanding of transformations and vectors from Chapter 34, along with your experience using the vector graphics API from Chapter 35, "Programming Vector Graphics," and Pixel Bender from Chapter 38, "Writing Shaders with Pixel Bender," to use the lowest-level 3D facilities provided by Flash Player 10. With these advanced 3D techniques, you can come much closer to drawing 3D solid models; you can even texture and light them. I have little doubt that the methods presented in this chapter were exposed to ActionScript 3.0 code solely to enable ActionScript developers to write custom software 3D rendering engines and to enable those existing software engines to move more of the processor-intensive tasks from ActionScript code to Flash Player.

The three key techniques covered here are projecting batches of points, drawing triangle strips, and UV-mapping textures. I also touch on techniques for backface culling, lighting, and shading. Use these techniques if you're writing a custom 3D renderer. If you're just looking to transform flat sprites, stick with the 3D display list API, and if you're more interested in using an engine than writing one, use one of the software engines covered in Chapter 15. For the brave and the curious, read on.

FEATURED CLASSES

```
flash.display.Graphics

flash.display
  .GraphicsTrianglePath

flash.geom.Utils3D

flash.geom
  .PerspectiveProjection

flash.geom.Matrix3D

flash.geom.Vector3D
```

Version

FP10. This entire chapter is applicable to Flash Player 10 and later. ∎

Game Plan

In this chapter, you'll start where prior chapters left off and build toward a more complete 3D engine that will display solid, textured, and lit models. You'll take this one step at a time:

1. You'll take a batch of vertices in 3D space as `Vector3D`s and project them all, using `Utils3D`.

2. You'll draw these as points using methods of `Graphics` that you already know.

3. You'll connect the vertices into a triangle strip and draw them using `drawTriangles()` or `drawGraphicsData()` with a `GraphicsTrianglePath`.

4. You'll experiment with wireframe drawing and solid fills.

5. You'll learn about backface culling and enable it so that you don't have to worry about overlapping polygons on convex objects.

6. You'll learn how a texture is mapped onto a model using UV mapping and map a simple texture on a model.

7. Finally, you'll get a taste of lighting and shading techniques using multipass textures and Pixel Bender shaders.

It's an ambitious plan, but if you understand matrices and vectors, you're already halfway there. (If you don't, please do try some of the 3D texts and sites I've cross-referenced in Chapter 34 before you go on.)

Projecting Batches of Points

In almost all renderers, solid 3D models are drawn as a continuous shell made up of many flat shapes — if you put the camera inside a model you'll see that it's empty. By breaking up what could be a curvy, complex surface into small pieces, you can draw something that appears to have depth, surface detail, even curves (one hard problem) by instead rendering thousands of little flat shapes with no depth (lots of easy problems). Again, in almost all 3D renderers, these objects are triangles. These polygons only *approximate* a complex surface; use more to get more detail, or use fewer to render faster.

Each triangle is a *face* of the object, and each point on the triangle is a *vertex*. Because a continuous surface is tessellated into many adjoining triangles, these vertices are shared between multiple triangles. But wait, let me explain more before going on.

First, according to plan, create a batch of points that make up a surface. As a simple example, I'll use an implicit surface defined by a 3D sine wave, shown in Example 40-1. It should look like a big ripple. It makes the most sense to represent a set of vertices as a `Vector` (in the `Array` sense) of `Vector3D`s (in the `Point` sense). You'll soon see that this isn't the only option, but I'll start with this.

After creating the set of points, you'll set up an animation that projects all the points into 2D and draws them.

EXAMPLE 40-1 `http://actionscriptbible.com/ch40/ex1`

Projecting Points One at a Time

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  public class ch40ex1 extends Sprite {
    protected var perspective:PerspectiveProjection;
    protected var viewMatrix:Matrix3D;
    protected var modelMatrix:Matrix3D;
    protected var model:Plot3D;
    public function ch40ex1() {
      stage.quality = StageQuality.MEDIUM;
      //center coordinates
      this.x = stage.stageWidth/2;
      this.y = stage.stageHeight/2;
      //set up "camera" perspective projection
      perspective = new PerspectiveProjection();
      perspective.fieldOfView = 50;
      perspective.projectionCenter = new Point(0, 0);
      //set up view projection
      //move camera up, back and angle down before projecting
      viewMatrix = new Matrix3D();
      viewMatrix.appendTranslation(0, 4, 11);
      viewMatrix.appendRotation(26, Vector3D.X_AXIS);
      viewMatrix.append(perspective.toMatrix3D());
      //set up model transformation
      modelMatrix = new Matrix3D();
      //set up model
      model = new Plot3D();
      model.plot(new Rectangle(-3, -3, 6, 6), 70);
      //render loop
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      //rotate model progressively
      modelMatrix.appendRotation(2, Vector3D.Y_AXIS);
      //when you project, first apply the model matrix and then the
      //view/projection matrix. Precalculate into one matrix here.
      var concatenatedMatrix:Matrix3D = modelMatrix.clone();
      concatenatedMatrix.append(viewMatrix);

      var v:Vector3D;
      graphics.clear();
      graphics.beginFill(0, 1);
      for each (v in model.vertices) {
        v = Utils3D.projectVector(concatenatedMatrix, v);
        graphics.drawRect(v.x, v.y, 1, 1);
      }
```

continued

EXAMPLE 40-1 *(continued)*

```
      }
    }
  }
  import flash.geom.Rectangle;
  import flash.geom.Vector3D;
  class Plot3D {
    public var vertices:Vector.<Vector3D>;
    public var tOffset:Number = 0;
    public function plot(xzBounds:Rectangle, resolution:Number = 200):void {
      vertices = new Vector.<Vector3D>();
      var r:Rectangle = xzBounds;
      for (var z:Number = r.top; z < r.bottom; z += r.height/resolution) {
        for (var x:Number = r.left; x < r.right; x += r.width/resolution) {
          var y:Number = Math.sin(Math.pow(x, 2) + Math.pow(z, 2) + tOffset);
          vertices.push(new Vector3D(x, y, z));
        }
      }
    }
  }
```

The Plot3D class is built by evaluating the 3D equation at regular intervals, which become the vertices of the model. The essential part of this example is the `for..each` loop that's been highlighted. It takes every vertex, translates it into screen space by projecting it with a perspective projection matrix, and draws the point in screen space. The `Utils3D` class contains two static methods for projecting vectors. The syntax of `projectVector()` here should be self-evident: it takes a projection matrix and a 3D point and projects the point into 2D space by using that matrix. Another new bit of code is the `toMatrix3D()` method of `PerspectiveProjection`. You know that a perspective projection matrix is just another matrix; Flash Player exposes it as a different class so that you can set properties of the perspective projection easily. The `toMatrix3D()` method just retrieves its internal matrix representation.

I threw some bonus ideas into this example. The idea of model and view transformations is ubiquitous. Model transformations move objects around the scene — in this case, spin the plot around the Y axis. View transformations move the viewport — here, moving your eyes back from the origin so you can see the plot, and up so you can look down onto it rather than viewing it front-on. Combine view transformations with a perspective projection, and you finally have the missing pieces of a real camera that can tilt and move around. Remember that the cool thing about transformation matrices is that their effects are chained by simple concatenation (matrix multiplication), an effect that I take advantage of here, first to create a `viewMatrix` that moves the camera and applies perspective, and then to concatenate the model and view matrices into `concatenatedMatrix` so only one matrix is needed when projecting each vertex. The order you apply matrices is essential; concatenate the `modelMatrix` after the `viewMatrix`, and you'd be rotating the flat, post-projection, image plane instead of rotating and projecting the model.

This method is pretty slow. You have to iterate through the vertices, projecting each one and drawing each one individually. But there are ways to project the vertices as a batch, as well as to batch up drawing commands. To take advantage of these, you have to start treating vertex lists a little bit more raw.

Instead of `Vectors` of `Vector3D` objects, store vertices in a `Vector` of `Numbers`; you and Flash Player have to trust each other that those numbers refer to components of vertices, ordered in the proper manner, which is to say, x_1,y_1,z_1,x_2,y_2,z_2, and so on. Once you store vertices this way, you can use the static `Utils3D` method `projectVectors()`, which projects a whole batch of points at once:

```
function projectVectors(projectionMatrix:Matrix3D,
                        vertices:Vector.<Number>,
                        projectedPoints:Vector.<Number>,
                        textureCoords:Vector.<Number>):void
```

The `projectionMatrix` should be familiar as the matrix used to project vertices in 3D space onto screen space. The `vertices` argument contains the original 3D vertices to project, flattened out into repeating sets of three x, y, and z coordinates as just described. The `projectedPoints` list, however, holds the result of the projection — you must send in a valid, non-null `Vector.<Number>`, but the method itself sets its value — this is an alternative to returning the value; notice that the return type is `null`. After the method is through, the `projectedPoints` list contains the vertices' position on the screen in 2D, as a list of coordinate components ordered x_1,y_1,x_2,y_2,x_3,y_3 and so on — with only two values per coordinate. Likewise, `textureCoords` is modified by the method. I'll explain how when I cover texture mapping; for now, you can send this parameter an empty but valid `Vector.<Number>`.

In Example 40-2, you use vertices in a flat layout to take advantage of `projectVectors()`. You also experiment with another, faster way of drawing the vertices.

EXAMPLE 40-2 `http://actionscriptbible.com/ch40/ex2`

Projecting Batches of Points, Drawing Batches of Strokes

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  public class ch40ex2 extends Sprite {
    protected var perspective:PerspectiveProjection;
    protected var viewMatrix:Matrix3D;
    protected var modelMatrix:Matrix3D;
    protected var model:Plot3D;

    protected var projectedPoints:Vector.<Number> = new Vector.<Number>();
    protected var texturePoints:Vector.<Number> = new Vector.<Number>();
    protected var drawCommands:Vector.<int>;
    public function ch40ex2() {
      stage.quality = StageQuality.MEDIUM;
      this.x = stage.stageWidth/2;
      this.y = stage.stageHeight/2;
      perspective = new PerspectiveProjection();
      perspective.fieldOfView = 50;
      perspective.projectionCenter = new Point(0, 0);
```

continued

EXAMPLE 40-2   *(continued)*

```
      viewMatrix = new Matrix3D();
      viewMatrix.appendTranslation(0, 4, 11);
      viewMatrix.appendRotation(26, Vector3D.X_AXIS);
      viewMatrix.append(perspective.toMatrix3D());
      modelMatrix = new Matrix3D();
      model = new Plot3D();
      model.plot(new Rectangle(-3, -3, 6, 6), 70);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      modelMatrix.appendRotation(2, Vector3D.Y_AXIS);
      var concatenatedMatrix:Matrix3D = modelMatrix.clone();
      concatenatedMatrix.append(viewMatrix);
      Utils3D.projectVectors(
        concatenatedMatrix, model.vertices, projectedPoints, texturePoints);
      graphics.clear();
      graphics.lineStyle(0, 1);
      if (!drawCommands) {
        drawCommands = new Vector.<int>();
        for (var i:int = 0; i < projectedPoints.length; i+=2) {
          drawCommands.push(((i/2) % model.resolution == 0)?
            GraphicsPathCommand.MOVE_TO : GraphicsPathCommand.LINE_TO);
        }
      }
      graphics.drawPath(drawCommands, projectedPoints);
    }
  }
}
import flash.geom.Rectangle;
import flash.geom.Vector3D;
class Plot3D {
  public var vertices:Vector.<Number>;
  public var tOffset:Number = 0;
  public var resolution:Number;
  public function plot(xzBounds:Rectangle, resolution:Number = 200):void {
    vertices = new Vector.<Number>();
    this.resolution = resolution;
    var r:Rectangle = xzBounds;
    for (var z:Number = r.top; z < r.bottom; z += r.height/resolution) {
      for (var x:Number = r.left; x < r.right; x += r.width/resolution) {
        var y:Number = Math.sin(Math.pow(x, 2) + Math.pow(z, 2) + tOffset);
        vertices.push(x, y, z);
      }
    }
  }
}
```
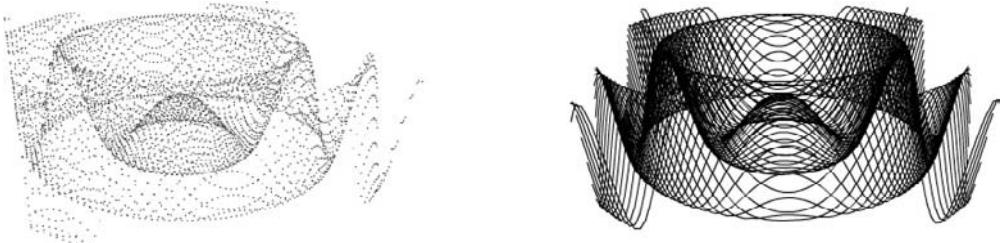
In the example, after `projectVectors()` fills up `projectedPoints` with the screen-space 2D points of the model, you batch up the drawing commands as well with `drawPath()`. If you remember from Chapter 35, `drawPath()` takes a list of commands and a list of points. You have the list of

points you'd like to draw, so all that remains is a set of drawing commands; here I've constructed one that connects all vertices at the same z value with a line. Finally, you pulled up all reusable `Vectors` to be instance variables, so that they don't have to be created anew every frame, an easy and effective optimization. Figure 40-1 shows the results for both Example 40-1 and Example 40-2.

FIGURE 40-1

Output from Examples

The optimizations in Example 40-2 should result in more fluid animation and reduced CPU utilization. Always batch up drawing commands and vector projection when you can.

Triangle Strips

Your 3D engine is well underway. You can create a set of vertices and draw them as points or a rudimentary kind of wireframe. The next stage is drawing something solid. To move from a network of points to a solid surface, you have to stop thinking about points and edges and start thinking about faces.

Vertices, edges, and faces are defined by the same data. Connect the points with lines, and you get edges. Fill in the triangles defined by three points, and you get faces. Figure 40-2 should help visualize this. You have all the points you need — but you need to help Flash Player tell which points define which triangles. Connecting the wrong vertices to form faces is a sure way to make a model useless.

No matter how you fill them, you draw solid faces (triangles) with the drawing API's `drawTriangles()` method — or its equivalent command object version, `GraphicsTrianglePath`. I'll just cover the imperative method; the objective one is nearly identical. The method, defined in `Graphics`, looks like this:

```
function drawTriangles(vertices:Vector.<Number>,
                       indices:Vector.<int> = null,
                       uvtData:Vector.<Number> = null,
                       culling:String = "none"):void
```
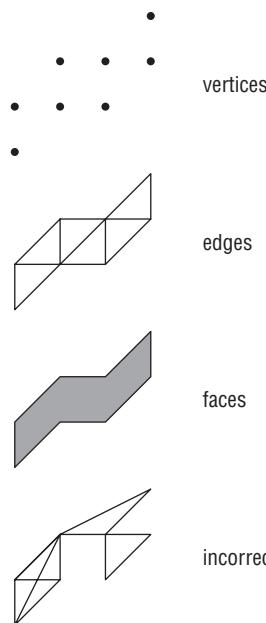
The `vertices` are the projected, 2D vertices, in a flat, xyxy order. You already have this data. The one missing piece, `indices`, tells Flash Player which points to connect into faces. Imagine that all vertices are assigned a number as they are inserted into the `vertices` list. You connect the dots by number: a single triangle is defined by three entries in the indices list, each referencing a vertex by its order. The `indices` are also stored flat and repeating, in sets of three. In this manner, you can define exactly how the triangles are constructed by the vertices. Figure 40-3 illustrates a sequence of vertices and the `indices` necessary to create triangles out of them.
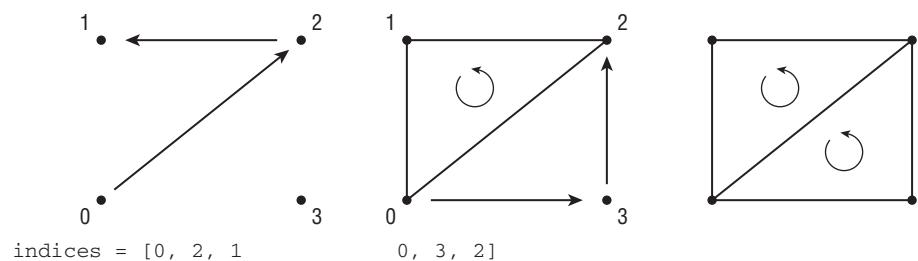
FIGURE 40-2

Vertices, edges, and faces. Vertices connected in the wrong order.

vertices

edges

faces

incorrect edges

FIGURE 40-3

Ordered vertices, and a sequence of indices that constructs triangles

```
indices = [0, 2, 1      0, 3, 2]
```

In almost every case, you'll mention vertices multiple times, as adjoining triangles share two vertices. Most times you'll be building up these layers of tessellated triangles in strips, almost like laying down bricks. These patterns can make programmatically creating the `indices` list easier. You *can* leave the indices vector empty, but this causes Flash Player to create one triangle out of every three vertices, losing its contiguous nature. This is rarely the right thing to do.

In Example 40-3, I'll start over with a trivially simple model so that you can picture exactly where each vertex is in your mind. A plane is about as simple as you can get, with four vertices and two triangles. I'll order the vertices and index the triangles exactly as pictured in Figure 40-3.

EXAMPLE 40-3 `http://actionscriptbible.com/ch40/ex3`

Drawing Triangles

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  import flash.utils.getTimer;
  public class ch40ex3 extends Sprite {
    protected var viewMatrix:Matrix3D;
    protected var modelMatrix:Matrix3D;
    protected var model:Plane3D;
    protected var projectedPoints:Vector.<Number> = new Vector.<Number>();
    protected var texturePoints:Vector.<Number> = new Vector.<Number>();
    public function ch40ex3() {
      this.x = stage.stageWidth/2;
      this.y = stage.stageHeight/2;
      var perspective:PerspectiveProjection = new PerspectiveProjection();
      perspective.fieldOfView = 50;
      perspective.projectionCenter = new Point(0, 0);
      viewMatrix = new Matrix3D();
      viewMatrix.appendTranslation(0, 0, 20);
      viewMatrix.append(perspective.toMatrix3D());
      modelMatrix = new Matrix3D();
      model = new Plane3D(new Rectangle(-8, -4.5, 16, 9));
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      modelMatrix.identity();
      modelMatrix.appendRotation(getTimer()/30, Vector3D.Y_AXIS);
      modelMatrix.appendRotation(45*Math.sin(getTimer()/900), Vector3D.X_AXIS);
      var concatenatedMatrix:Matrix3D = modelMatrix.clone();
      concatenatedMatrix.append(viewMatrix);
      Utils3D.projectVectors(
        concatenatedMatrix, model.vertices, projectedPoints, texturePoints);
      graphics.clear();
      graphics.lineStyle(0, 1, 0.5);
      graphics.beginFill(0x808080, 1);
      graphics.drawTriangles(projectedPoints, model.indices);
    }
  }
}
import flash.geom.Rectangle;
import flash.geom.Vector3D;
class Plane3D {
  public var rect:Rectangle;
  public var vertices:Vector.<Number>;
  public var indices:Vector.<int>;
  public function Plane3D(r:Rectangle):void {
    rect = r;
```

continued

EXAMPLE 40-3 *(continued)*

```
    vertices = new Vector.<Number>();
    indices = new Vector.<int>();
    vertices.push(
      r.left, r.bottom, 0, //bottom left = 0
      r.left, r.top, 0,  //top left = 1
      r.right, r.top, 0, //top right = 2
      r.right, r.bottom, 0 //bottom right = 3
    );
    indices.push(
      0, 2, 1, //left-side triangle BL->TR->TL
      0, 3, 2 //right-side triangle BL->BR->TR
    );
  }
}
```

You can play with the fill and stroke styles for various effects. This example demonstrates both a wireframe and a fill. For realistic rendering, you don't want to expose the vertices and edges, and you'll keep strokes turned off.

Make sure that you can follow along with the way that the triangles were created, because there's one more thing to worry about. Notice how I ordered the vertices in each triangle. To you or me, a triangle is the same no matter what order the vertices come in. But to a 3D engine, the direction in which the vertices of a face are defined — called the *winding order* — encodes some important information about the face. Imagine you're looking at a piece of glass, and an assistant has drawn an arrow around the edge going clockwise. Now he turns the glass around so that it faces away from you. The glass looks the same from the other side, but you notice that the arrow is pointing counterclockwise. The direction that arrow goes indicates which direction the glass is facing. (You can try this yourself with just a piece of paper. Draw an arrow going in a clockwise circle, flip the paper over, and hold it up to the light.)

In the same way, if you know for certain that you defined every face with the points going clockwise, when you see those points going counterclockwise, you know that part of the object is facing away from you. In the plane example, notice that both triangles were intentionally defined counterclockwise. It doesn't matter much which direction you define the points of a triangle, as long as you're consistent.

Backface Culling

When you're rendering a convex object by drawing its shell (broken up into lots of little polygons), there's one problem. Think of a simple object like a sphere. The shell of the sphere goes all the way around the sphere, of course. When you look at the sphere, however, you only see part of the shell. The back side is right there on the far side of the sphere, but if the sphere is opaque like a baseball, you can't see through to it. One way to solve this problem is to z-sort the polygons, drawing the front of the shell on top of the back, so that all you see is the opaque front side. However, `drawTriangles()` is drawing all the triangles on the same `Graphics` object at the same depth.

(I'll show how to z-sort faces in the section "Z-Sorting, Shading, and Further Topics.") Another approach is to determine which part of the shell faces away from you and simply elect not to draw it. So in reality only the front half of the baseball would exist, but you'd never know. As you rotate the ball, the new parts that face you become visible. You can never see parts of it that face away from you. (If the object isn't convex, this guarantee breaks down.)

When you don't draw polygons that face away from you, those polygons are sometimes called single-sided; removing those faces from the list of polygons to draw is called *backface culling*. Beyond solving the depth sorting issue for convex objects, it's an easy way to speed up rendering, because (to make a rash generalization) around half of the surface is usually facing away from you.

To enable backface culling, first make sure that all your triangles are defined in the same winding order. Then pass an appropriate culling mode to the `culling` parameter of `drawTriangles()`: `TriangleCulling.POSITIVE` to remove back-facing polygons when you define the faces in a counterclockwise winding order, `TriangleCulling.NEGATIVE` when they're defined in a clock-wise winding order. Or, use the opposite value to cull front-facing polygons for whatever reason. Passing in `TriangleCulling.NONE` turns off backface culling, rendering all polygons (the default). If you modify the `drawTriangles()` call in Example 40-3 to read

```
graphics.drawTriangles(projectedPoints, model.indices,
                       null, TriangleCulling.POSITIVE);
```

you notice that the plane disappears for half its revolution. Later in the chapter, you apply backface culling to a more exciting shape and see that it really works.

Texture Mapping

Your 3D engine can now display solid convex meshes without overlapping the far side of the object, but something's still lacking. You're limited to the fill and line styles that `Graphics` provides; your models are drawn with a flat color over the entire thing. Even if you picked a particularly appealing color, it's going to look like a flood filled silhouette, completely flat, mocking your attempts at realism. Mocking!

The next evolution in your increasingly impressive 3D engine is textures. Texture mapping takes a single flat image and uses it to paint a whole set of 3D faces. Basically, each triangle in the model gets assigned a triangle-shaped area in the image. When the triangle is drawn, instead of being filled with a solid color, it's filled with the part of the texture map it's associated with. Now, if that triangle appears at an angle to the viewer, the texture triangle is distorted to match; this is how texture-mapped objects give the impression of depth.

You can do an impressive amount with a simple texture map, because you're free to design it down to the pixel. Faces define the shape of the model; a texture map defines its appearance.

You define the mapping between triangles on a 3D surface and areas on a texture by adding another set of data to the vertices. Each vertex in 3D is mapped to a point on the texture image. This is also called *UV-mapping* because you traditionally use another set of coordinates to refer to texture-space: u and v (and sometimes t), so you map XYZ coordinates into UV coordinates.

The coordinate system of texture-space is really simple (forgetting about t). Much like screen-space, the top-left is the origin (0, 0), u increases to the right, and v increases down. The entire texture is always scaled to exist within (0, 0) to (1, 1), or you can think of it the other way: that space is scaled

to fit a unit square to the size of the texture. One way or another, the u and v coordinates indicate a percentage of the full width and height of the texture, so that (0.5, 0.5) is the center of the image.

Let's start by texture mapping a simple object, the plane. Mapping a texture onto a plane is a degenerate example, because you just map each corner to each corner, as Figure 40-4 shows. Doing so by hand and visually (in Example 40-4) helps you understand the mapping, however. Once you get the example running, you can see the effect of changing the UV map.

EXAMPLE 40-4 http://actionscriptbible.com/ch40/ex4

Texture Mapping a Plane

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  import flash.utils.getTimer;
  public class ch40ex4 extends Sprite {
    protected var viewMatrix:Matrix3D;
    protected var modelMatrix:Matrix3D;
    protected var model:Plane3D;
    protected var texture:BitmapData;
    protected var projectedPoints:Vector.<Number> = new Vector.<Number>();
    public function ch40ex4() {
      this.x = stage.stageWidth/2;
      this.y = stage.stageHeight/2;
      var perspective:PerspectiveProjection = new PerspectiveProjection();
      perspective.fieldOfView = 50;
      perspective.projectionCenter = new Point(0, 0);
      viewMatrix = new Matrix3D();
      viewMatrix.appendTranslation(0, 0, 20);
      viewMatrix.append(perspective.toMatrix3D());
      modelMatrix = new Matrix3D();
      model = new Plane3D(new Rectangle(-8, -4.5, 16, 9));
      var l:Loader = new Loader();
      l.load(new URLRequest(
        "http://actionscriptbible.com/files/texture-approved.jpg"),
        new LoaderContext(true));
      l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    }
    protected function onLoad(event:Event):void {
      texture = Bitmap(LoaderInfo(event.target).content).bitmapData;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      modelMatrix.identity();
      modelMatrix.appendRotation(45*Math.cos(getTimer()/582), Vector3D.Y_AXIS);
      modelMatrix.appendRotation(45*Math.sin(getTimer()/900), Vector3D.X_AXIS);
      var concatenatedMatrix:Matrix3D = modelMatrix.clone();
      concatenatedMatrix.append(viewMatrix);
```

```
      Utils3D.projectVectors(
        concatenatedMatrix, model.vertices, projectedPoints, model.uvt);
      graphics.clear();
      graphics.beginBitmapFill(texture, null, false, true);
      graphics.drawTriangles(projectedPoints, model.indices, model.uvt);
    }
  }
}
import flash.geom.Rectangle;
import flash.geom.Vector3D;
class Plane3D {
  public var vertices:Vector.<Number> = new Vector.<Number>();
  public var uvt:Vector.<Number> = new Vector.<Number>();
  public var indices:Vector.<int> = new Vector.<int>();
  public function Plane3D(r:Rectangle):void {
    vertices.push(
      r.left, r.bottom, 0, //bottom left = 0
      r.left, r.top, 0,  //top left = 1
      r.right, r.top, 0, //top right = 2
      r.right, r.bottom, 0 //bottom right = 3
    );
    uvt.push(
      0, 1, 0, //bottom left
      0, 0, 0, //top left
      1, 0, 0, //top right
      1, 1, 0 //bottom right
    );
    indices.push(
      0, 2, 1, //left-side triangle BL->TR->TL
      0, 3, 2 //right-side triangle BL->BR->TR
    );
  }
}
```

This doesn't seem like much of an achievement, because you've essentially just re-created what you could do by setting the rotationX property of an image. But now that you understand how it's done, you can easily apply these techniques to bigger and better models!

The UV map is set by assigning UVT coordinates to the uvt Vector, again in a flat list of Numbers in uvtuvt order. Set the t coordinate to 0. When you project the vertices, notice that the UV map is sent as well; when Utils3D projects the vertices, it remembers how far away each one is and tucks that away in the t coordinates of the UV map, modifying it inline. That's why you pass uvt to projectVectors() as well. (Because it doesn't touch the other data, you can let it reuse that Vector, overwriting old values for t every time it projects.) The t coordinate, because it stores distance, helps distort the texture triangle in perspective.
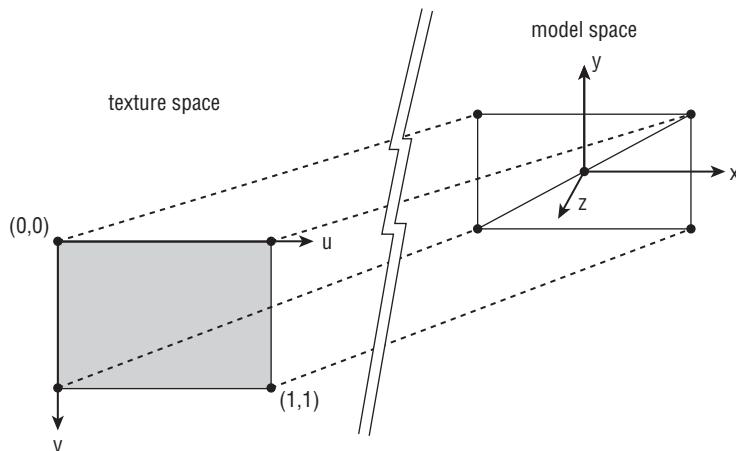
Finally, you use a bitmapFill() and pass in the UV coordinates to drawTriangles(). When Graphics sees that you're using a bitmap fill and you've provided this vector, it takes over and maps the texture to each triangle; otherwise, the bitmap is used as a normal fill. You can still set properties of the bitmapFill() — turn on smoothing to make the texture look a bit nicer.

867

FIGURE 40-4

Texture space, model space, and the UV mapping between them. For a flat surface, the mapping is direct.

In Example 40-5, you texture map a much more complex object, using the same techniques. The result is shown in Figure 40-5.

EXAMPLE 40-5 http://actionscriptbible.com/ch40/ex5

A Tasty Textured Torus

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  [SWF(backgroundColor="#B0E1E3")]
  public class ch40ex5 extends Sprite {
    protected var viewMatrix:Matrix3D;
    protected var modelMatrix:Matrix3D;
    protected var model:Torus3D;
    protected var texture:BitmapData;
    protected var projectedPoints:Vector.<Number> = new Vector.<Number>();
    public function ch40ex5() {
      stage.quality = StageQuality.LOW;
      this.x = stage.stageWidth/2;
      this.y = stage.stageHeight/2;
      var perspective:PerspectiveProjection = new PerspectiveProjection();
      perspective.fieldOfView = 80;
      perspective.projectionCenter = new Point(0, 0);
```

```
      viewMatrix = new Matrix3D();
      viewMatrix.appendTranslation(0, 7, 12);
      viewMatrix.appendRotation(40, Vector3D.X_AXIS);
      viewMatrix.append(perspective.toMatrix3D());
      modelMatrix = new Matrix3D();
      model = new Torus3D(2, 6, 40, 40);
      var l:Loader = new Loader();
      l.load(new URLRequest(
        "http://actionscriptbible.com/files/texture-donut.jpg"),
        new LoaderContext(true));
      l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    }
    protected function onLoad(event:Event):void {
      texture = Bitmap(LoaderInfo(event.target).content).bitmapData;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      modelMatrix.appendRotation(2, Vector3D.Y_AXIS);
      var concatenatedMatrix:Matrix3D = modelMatrix.clone();
      concatenatedMatrix.append(viewMatrix);
      Utils3D.projectVectors(
        concatenatedMatrix, model.vertices, projectedPoints, model.uvt);
      graphics.clear();
      graphics.beginBitmapFill(texture, null, false, false);
      graphics.drawTriangles(
        projectedPoints, model.indices, model.uvt, TriangleCulling.POSITIVE);
    }
  }
}
import flash.geom.Rectangle;
import flash.geom.Vector3D;
class Torus3D {
  public var vertices:Vector.<Number> = new Vector.<Number>();
  public var uvt:Vector.<Number> = new Vector.<Number>();
  public var indices:Vector.<int> = new Vector.<int>();
  public function Torus3D(crossSectionRadius:Number, radiusToTube:Number,
                          uResolution:Number = 50, vResolution:Number = 50) {
    var R:Number = radiusToTube, r:Number = crossSectionRadius;
    var uStep:Number = Math.PI * 2 / uResolution;
    var vStep:Number = Math.PI * 2 / vResolution;
    var FLOAT_ERROR:Number = 0.0001;
    for (var u:Number = 0; u <= Math.PI*2 + FLOAT_ERROR; u += uStep) {
      for (var v:Number = 0; v <= Math.PI*2 + FLOAT_ERROR; v += vStep) {
        var x:Number = (R + r * Math.cos(v)) * Math.cos(u);
        var y:Number = r * Math.sin(v);
        var z:Number = (R + r * Math.cos(v)) * Math.sin(u);
        vertices.push(x, y, z);
        uvt.push(u / (Math.PI*2), v / (Math.PI*2), 0);
      }
    }
```

EXAMPLE 40-5 *(continued)*

```
    for (var ui:int = 0; ui <= uResolution; ui++) {
      for (var vi:int = 0; vi <= vResolution; vi++) {
        var thisSlice:int = ui * vResolution + vi;
        var nextSlice:int = (ui+1) * vResolution + vi;
        indices.push(thisSlice, nextSlice + 1, nextSlice,
          thisSlice, thisSlice + 1, nextSlice + 1);
      }
    }
  }
}
```

FIGURE 40-5

How a 2D texture is mapped onto a more interesting 3D model. The donut is the output of
Example 40-5.

Try running the example. I don't know about you, but I totally headed to Krispy Kreme after writing
this. See if you can follow along with how the texture is mapped both in this example and in code.
The UV map is simple because the object (a torus) is created as a function of u and v to begin with.
If you want to see how the triangles are created, try turning the model resolution lower and drawing
lines in addition to the texture. Essentially, I stepped through u and v, connecting every set of four
points with two triangles, making sure that all of them were in the same order. You can see that this
works correctly with backface culling in the example — but because the torus isn't totally convex,
there are camera angles that show some overlapping areas of the model. I've carefully chosen a projec-
tion that avoids this. Finally, you have to mind the gap — the gap between the end of the model and
the beginning. If you construct this incorrectly, you can end up with a gap in the torus (as my first
attempt did), or with the texture mapped incorrectly in the final segment (as my second attempt did).
This is the purpose of FLOAT_ERROR: to make sure that a final set of vertices is created that overlaps
the first one (at 0 and 2π), avoiding any gaps.

Z-Sorting, Shading, and Further Topics

You've learned a huge amount about 3D graphics in a small space and put the essentials in place for
a decent 3D engine. In this final section, I'll speed up a bit and introduce some topics for further
research.

Polygon Z-Sorting

In "Backface Culling," you saw that backface culling removes faces that point away from you, hiding what would otherwise be overlapping portions of convex objects. However, even an object as simple as the torus in Example 40-5 overlaps itself in places despite backface culling, and more complex objects are likely to have problems as well. Backface culling is still a great optimization, because figuring out that a polygon is facing away takes far, far less time than drawing it to the screen, but the real solution to overlapping polygons is to sort them by their distance from the viewport.

Yes, all triangle drawing may take place on a single display object, but you can change the order in which your triangles are drawn. Draw the bottommost triangles first and the topmost triangles last to get the proper effect, just like the 2D painter's algorithm introduced in Chapter 14, "Visual Programming with the Display List." Even if some triangles overlap, the topmost ones are visible.

The texture coordinate t is calculated during projectVectors() and stores the distance to the vertex post-projection. So you can reuse this value to sort the list of triangles. In Example 40-6, you add z-sorting to the faces of the donut and rotate it to see that no two faces overlap. The changes are in bold.

EXAMPLE 40-6   http://actionscriptbible.com/ch40/ex6

Z-Sorting Faces

```
package {
  import flash.display.*;
  import flash.events.Event;
  import flash.geom.*;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;
  [SWF(backgroundColor="#B0E1E3")]
  public class ch40ex6 extends Sprite {
    protected var viewMatrix:Matrix3D;
    protected var modelMatrix:Matrix3D;
    protected var model:Torus3D;
    protected var texture:BitmapData;
    protected var projectedPoints:Vector.<Number> = new Vector.<Number>();
    public function ch40ex6() {
      stage.quality = StageQuality.LOW;
      this.x = stage.stageWidth/2;
      this.y = stage.stageHeight/2;
      var perspective:PerspectiveProjection = new PerspectiveProjection();
      perspective.fieldOfView = 80;
      perspective.projectionCenter = new Point(0, 0);
      viewMatrix = new Matrix3D();
      viewMatrix.appendTranslation(0, 7, 12);
      viewMatrix.appendRotation(40, Vector3D.X_AXIS);
      viewMatrix.append(perspective.toMatrix3D());
      modelMatrix = new Matrix3D();
      model = new Torus3D(2, 4, 20, 20);
      var l:Loader = new Loader();
```

continued

EXAMPLE 40-6   *(continued)*

```
      l.load(new URLRequest(
        "http://actionscriptbible.com/files/texture-donut.jpg"),
        new LoaderContext(true));
      l.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
    }
    protected function onLoad(event:Event):void {
      texture = Bitmap(LoaderInfo(event.target).content).bitmapData;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    protected function onEnterFrame(event:Event):void {
      modelMatrix.appendRotation(-0.4, Vector3D.Z_AXIS);
      modelMatrix.appendRotation(0.8, Vector3D.X_AXIS);
      modelMatrix.appendRotation(2, Vector3D.Y_AXIS);
      var concatenatedMatrix:Matrix3D = modelMatrix.clone();
      concatenatedMatrix.append(viewMatrix);
      Utils3D.projectVectors(
        concatenatedMatrix, model.vertices, projectedPoints, model.uvt);
      graphics.clear();
      graphics.beginBitmapFill(texture, null, false, false);
      graphics.drawTriangles(projectedPoints, model.getZSortedIndices(),
        model.uvt, TriangleCulling.POSITIVE);
    }
  }
}
import flash.geom.Vector3D;
class Torus3D {
  public var vertices:Vector.<Number> = new Vector.<Number>();
  public var uvt:Vector.<Number> = new Vector.<Number>();
  public var indices:Vector.<int> = new Vector.<int>();
  private var zSort:Vector.<ZSort>;
  public function Torus3D(crossSectionRadius:Number, radiusToTube:Number,
                          uResolution:Number = 50, vResolution:Number = 50) {
    var R:Number = radiusToTube, r:Number = crossSectionRadius;
    var uStep:Number = Math.PI * 2 / uResolution;
    var vStep:Number = Math.PI * 2 / vResolution;
    var FLOAT_ERROR:Number = 0.0001;
    for (var u:Number = 0; u <= Math.PI*2 + FLOAT_ERROR; u += uStep) {
      for (var v:Number = 0; v <= Math.PI*2 + FLOAT_ERROR; v += vStep) {
        var x:Number = (R + r * Math.cos(v)) * Math.cos(u);
        var y:Number = r * Math.sin(v);
        var z:Number = (R + r * Math.cos(v)) * Math.sin(u);
        vertices.push(x, y, z);
        uvt.push(u / (Math.PI*2), v / (Math.PI*2), 0);
      }
    }
    zSort = new Vector.<ZSort>();
    for (var ui:int = 0; ui <= uResolution; ui++) {
      for (var vi:int = 0; vi <= vResolution; vi++) {
        var thisSlice:int = ui * vResolution + vi;
        var nextSlice:int = (ui+1) * vResolution + vi;
        if ((nextSlice + 1) * 3 + 2 < vertices.length) {
```

```
          indices.push(thisSlice, nextSlice + 1, nextSlice,
            thisSlice, thisSlice + 1, nextSlice + 1);
          zSort.push(new ZSort(thisSlice, nextSlice + 1, nextSlice),
            new ZSort(thisSlice, thisSlice + 1, nextSlice + 1));
        }
      }
    }
  }
  public function getZSortedIndices():Vector.<int> {
    var node:ZSort;
    for each (node in zSort) {
      node.t = Math.min(uvt[node.i1*3+2], uvt[node.i2*3+2], uvt[node.i3*3+2]);
    }
    zSort = zSort.sort(ZSort.compare);
    var zSortedIndices:Vector.<int> = new Vector.<int>();
    for each (node in zSort) zSortedIndices.push(node.i1, node.i2, node.i3);
    return zSortedIndices;
  }
}
class ZSort {
  public var i1:int, i2:int, i3:int, t:Number;
  public function ZSort(i1:int, i2:int, i3:int) {
    this.i1 = i1; this.i2 = i2; this.i3 = i3;
  }
  public static function compare(a:ZSort, b:ZSort):int {
    return (a.t < b.t)?-1:1;
  }
}
```

For each face, the minimum projected depth of the three vertices is used as the face's depth.

Shading and Lighting Introduced

Things look pretty good, but still flat, because you're not taking any light into account. It's as if every point of the donut is being hit with equal light from every direction. To increase realism, the next step is setting up light sources and shading the model.

First consider a single, directional light source. It casts light rays at a certain angle, all parallel to each other. You can represent this kind of light as a vector pointing in the correct angle. Besides ambient light — consistent light from every angle, which can be achieved simply by dimming or brightening the texture — this is the simplest form of lighting. When you combine ambient light with directional light, it's a rough approximation of direct sunlight: some parts of the sun's light are bounced around more or less randomly, to hit the object more or less evenly, and the rest directly hit the object from the angle of the sun in the sky.

Adding a light to the scene is all well and good, but you only notice its presence when it interacts with objects in the scene. When you draw the surface of an object, changing the colors to make some impression — like that of light, or surface texture, or reflectivity — you're *shading* an object.

In many 3D engines, the simplest kinds of lighting, and the first you'll learn, are per-face shading (like flat shading) and per-vertex shading (like Gourad shading). In Flash Player, you're not drawing every

873

polygon individually. That approach is too slow. Instead, you draw the entire object with a bitmap that's UV-mapped. You can't specify a fill per face or a fill per vertex. Instead, you use a more sophisticated technique: you modify the texture so that the texture itself looks lit correctly.

You can do this with some simple effects — a `ColorTransformFilter` will be sufficient to light a model with one or more directional and ambient lights — or with a Pixel Bender shader for even more complex (and optimized) effects such as reflectivity, environment mapping, bump mapping, normal mapping, and so on. Whichever approach you take, because you're taking a texture and modifying it at least once, this can be called *multitexturing* (or a *multipass texture*).

To calculate per-pixel lighting, you need to generate a *normal map*.

Normal Maps

A *normal* is a vector that points directly out of a face, perpendicular to its surface. The way light affects the appearance of a surface depends on the angle of the light and the angle of the surface. So normals are key to calculating lighting.

Because you have all the information on the geometry of an object, you can determine its normals mathematically. (Take the cross product of the vectors that define two edges of the triangle.) This approach is useful when you're doing per-face or per-vertex lighting; you can calculate the normals dynamically and on-demand. You can do so with more detail *and* faster by precalculating the normals for every pixel on the surface. A *normal map* uses the same UV coordinates and UV mapping as the texture map, but instead of a color, it stores a normal at every pixel. One way to do this is to encode the X, Y, and Z components of an Euler angle in the R, G, and B channels. Or you can encode the four components of a quaternion in R, G, B, and A.

In 3D models generated with a modeling package, you'll usually be able to build a normal map directly in software. With a primitive like a torus, it's easily calculated.

Note

An awesome side effect of normal mapping is that the normal map can contain far more detail than the 3D model itself. For instance, you can create a head in a 3D modeling package that has 3 million polygons, render a normal map using this geometry, and then simplify the geometry down to 800 polygons. So in your real-time 3D engine, you only draw 800 faces (or the subset of these that face forward, naturally), but you shade the model per-pixel with the geometry of the original, complex model. ∎

Shading with a Normal Map

Once you have a normal map and a texture map, you can calculate all kinds of lighting effects.

- Bump mapping and simple lighting — Apply a `ColorTransformFilter` to the normal map that multiplies each RGB component (representing an XYZ component) with the light's XYZ components. This is an extremely streamlined method for getting the dot product of the normal vector and the light vector. Use the effected normal map to multiply the color values from the texture, and you get the effect of a single light on a surface. Apply multiple times for multiple light sources.

- Environment mapping — Reflective surfaces are usually rendered with the help of an environment map, which wraps a 360° image around the whole scene (usually without displaying said image). Then use the normal at each point to cast out a ray to the cube or sphere environment. Use the pixel in the environment map that this normal points at as the (or to influence the) color at the point on the model, and it reflects its environment. Using an additional image input for the environment map, you can implement this with a Pixel Blender filter.

- Cel shading — Also known as cartoon shading, you can posterize a texture after calculating lighting to reduce the number of colors used and give the model an interesting effect. Try using `threshold()` or `paletteMap()` in `BitmapData` to create the posterized effect.

And more. Modifying the texture map in relation to lights and the geometry of the object (encoded in its normal map), you can come up with all kinds of impressive 3D effects and optimize them to run in real time. There are plenty of impressive posts by Raph Hauwert (`http://unitzeroone.com/blog/`), Andy Zupko (`http://blog.zupko.info/`), David Lenaerts (`http://derschmale.com/`), Ricardo Cabello (`http://mrdoob.com/blog`), and others that show just how creative you can be with 3D effects, especially with the power of Pixel Bender.

Summary

- Flash Player 10 and later include low-level math utilities that facilitate 3D engines more powerful than the 2.5D `DisplayObject` API. The coding is up to you.

- Some of the software 3D libraries mentioned in Chapter 15 already take advantage of these, and they make displaying 3D much easier, if you're more interested in getting things on the screen.

- `Utils3D` contains two static methods used to project a single 3D vector and a batch of vectors; they take a perspective projection matrix.

- You can retrieve the transformation matrix from a `PerspectiveProjection` object through its `toMatrix3D()` method.

- 3D models are at minimum a list of vertices. To display them as solid faces, you need to order the vertices into triangles.

- Use the `drawTriangles()` method of `Graphics` to draw solid or wireframe surfaces to the screen.

- When using a bitmap fill with `drawTriangles()`, UV-mapped coordinates may be supplied; these fill each face with the appropriate piece of a texture.

- `Utils3D` also transforms UV coordinates.

- Backface culling skips drawing any part of the surface that faces away after projection. This is calculated by the winding order of the polygon, so make sure all triangles are defined in the same order.

- Sort the list of triangles by their projected z value in order before sending to `drawTriangles()`, and the polygons will be z-sorted.

- By constructing a normal map and modifying the texture based on the lights and normal at each point, you can shade objects per pixel.

Part IX

Flash in Context

Globalization, Accessibility, and Color Correction

W hy do these topics belong together? They are all tools you can use to ensure that your content is usable and beautiful for everyone, regardless of their primary language, sightedness, and monitor settings. In this chapter, you'll learn how to use all of these, and I'll briefly give mention to other user-adaptive technologies and techniques that have been covered in prior chapters.

Globalization and Localization

An important step in the planning process of any project is to identify what *locales* it will be deployed in. In this context, locale doesn't refer to a specific geographic area so much as a set of conventions: not just the language, but its dialect (U.S. English vs. British English); its measurement system (Metric/SI or Imperial); its currency, date, and numeric conventions (€7,00 vs. $7.00); and so on. Essentially, a locale is the set of regional rules surrounding written material. As you can imagine, supporting more than one locale means making some changes in the way you present text and numbers. Knowing the locales ahead of time, or at least knowing the fact that you need to support more than one locale, lets you write your user interface code with localization in mind and saves you from going back and ripping apart perfectly good, tested code later.

Now that I've dispensed with that obligatory warning, I'll return to those terms. *Globalization*, also called *internationalization*, entails writing code that's easy to localize. It's that planning ahead part I just mentioned. *Localization* is the process of adapting it to a specific locale. For example, this code has not been internationalized; it's not easily localized:

```
priceTF.text = "$" + priceValue.toFixed(2);
button.labelTF.text = "Add to Cart";
button.labelTF.width = 200;
```

There are specific problems with code like this; moreover, these problems are the same problems you'll see over and over again. The things you have to watch out for when localizing are almost always the same, including

- A specific, hardcoded algorithm being used to convert numeric quantities into text.
- User-facing text being hardcoded.
- Dimensions and positions linked to presumed size of text. This is a huge, often overlooked issue in localization; it frequently takes more or less space to say the same thing in different languages. (And don't get me started with the impact that vertical versus horizontal or left-to-right versus right-to-left text has on a layout!)

There are many solutions to these localization problems. Of course, the general approach is usually the same:

- Use code for outputting numeric quantities that can be parameterized by locale.
- Use some sort of string lookup table that can be parameterized by locale.
- Write liquid, dynamically sizing, and positioning user interfaces when possible.

Of course, remember that your first step should be to identify which locales your application will be used in. You can cut corners safely if you're assured that the application is limited to certain locales. Furthermore, it comes as no surprise that there are plenty of tools around to solve these well-known problems.

Rudimentary locale support exists as the `toLocaleString()` method of any `Object`. Almost zero classes, however, override this method for locale-specific formatting. Only `Date` seems to use it.

If you're using the Flex framework, for instance, there's a great `ResourceManager` that lets you replace text and even assets and data, by locale, at compile time or runtime. You can easily create resource bundles for each locale by editing properties files, which are very simple. The Flex framework also provides a set of `Formatter` objects that make it easy to reformat text for different locales. It's not feasible to use these classes outside a Flex project, so you may consider moving to Flex to get all these localization features "for free."

More to the point, ActionScript 3.0 has its own globalization API, found in the `flash.globalization` package. Found in Flash Player 10.1 and later, the API contains formatters for dates, times, currencies, and numbers; it can more intelligently compare and modify characters and strings than the `String` class. Better yet, regional support is built in; you don't have to code the rules yourself.

Version

FP10.1. All `flash.globalization` classes are available in Flash Player 10.1 and later. The remainder of this section covers these classes and is applicable only to this version. ∎

Together with the new Flash Text Engine and its great international capabilities, the globalization package in Flash Player 10.1 makes Flash Player truly a citizen of the world.

Identifying Locale

Locales are represented by a standardized code (following the spec at `http://unicode.org/reports/tr35/`) as a `String`. To simplify that document tremendously, a locale is uniquely identified by a language code and potentially a region. Each of these is two characters. For instance,

my primary region is `en-US`, which stands for English as it's used in the United States. I can also certainly read `en-GB` (British English) or to lesser extents `ja` (Japanese) and `es` (Spanish). A language without a region is still a valid locale. The `LocaleID` class helps standardize locale strings and provides you with a little information about a locale. It's not necessary to encapsulate the locale code in that class, though. You can keep it around as a string.

So this begs the question, how do you determine the user's locale? It's an interesting question that's not as easy to answer as you'd think. Although you can simply use the OS default, you should put some thought into the best way to identify the locale.

Using the Default That the Operating System Provides

The simplest method of choosing the user's preferred locale is listening to the operating system default. `LocaleID.DEFAULT` represents the OS's current locale. But it's a bit quirky; it isn't a static accessor that dynamically grabs the current locale string. It's actually a constant, special-case locale string: `i-default`. No code retrieves an actual locale until you try to use this locale with one of the classes in the globalization package. Only when it's used is the real locale retrieved.

The globalization classes have a common mechanism in which they're initialized with a locale, but they may end up using a different locale if that one's not available to the OS. These two locales — the one that's requested and the one that's resolved — are made available to ActionScript through the `actualLocaleIDName` and `requestedLocaleIDName` properties of the globalization classes. If you need to, then, you can resolve the actual locale by creating a globalization class instance.

```
var g11n:StringTools = new StringTools(LocaleID.DEFAULT);
trace("Requested locale =", g11n.requestedLocaleIDName); //i-default
trace("Actual locale =", g11n.actualLocaleIDName); //en-US (for me)
```

However, you'll usually be fine using `LocaleID.DEFAULT`.

Using the OS default locale is easy enough. Consider this, however. People may use their OS in the default language if it's difficult to configure, if it came preinstalled with a fixed language, or if foreign language packs cost extra. For instance, Windows 7 supports user accounts with different individual languages only in its more expensive Enterprise and Ultimate editions.

Based on Location

Can you use location to infer the user's locale? Not necessarily. I'm sure you know at least one person who lives in your country but doesn't speak the national language as his primary language. Maybe this even applies to you, your parents, or your grandparents. Using location to infer locale is a start, but it's not nearly accurate.

Based on the Browser

If your application is living on the web, you can ask the browser for its language. There may be a better chance that the user is using the correct-language edition of her browser, at least. Using JavaScript (and `ExternalInterface`, covered in Chapter 43, "Interfacing with JavaScript"), you can get this property from JavaScript's `navigator` object. The `language` property is usually what you want here, but it can go by several different names. However, this simply returns the language that the browser's UI is in.

Based on the Browser's Configuration

Perhaps the most useful way to get the locale is to find out what languages the browser is prepared to accept from the web. You can set this up in the preferences of your browser, specifying as many languages as you're comfortable reading, in whatever order you choose. The browser sends this information to web servers in the `Accept-Language` HTTP header so that the server can choose to prepare your content in the correct language. You can't get a handle on this request using Flash Player, but there's a simple workaround: make a request to a script that responds with the contents of that header. You request the page in ActionScript; the browser sends the HTTP request, including that header with the languages you can read; and the server, instead of using this hint to deliver the right language, spits out the whole list of languages verbatim, for ActionScript to then parse. Got it?

For instance, in PHP, you simply write a script with these contents:

```
<?php echo($_SERVER[HTTP_ACCEPT_LANGUAGE]); ?>
```

Put it on a web server with PHP, and try hitting it with your browser to see what languages your browser is configured to accept.

The benefit of this approach is that (ideally) you get a list of all acceptable locales, sorted by preference. In Figure 41-1, you can see the languages I've set in Firefox. The request string that this produces is

```
en-us,en;q=0.8,ja;q=0.5,es;q=0.3
```

FIGURE 41-1

Language settings in Firefox

Just Ask

Guess what? It may be annoying, but the only way to be absolutely sure the user is using the locale most comfortable to her is to ask her. I think the best technique is to guess intelligently using the techniques described earlier, but give the user the opportunity to change the locale manually.

An Example

In Example 41-1, you see two browser-based locale retrieval methods in action, and you see the information you can pull out of a locale with `LocaleID`.

EXAMPLE 41-1 `http://actionscriptbible.com/ch41/ex1`

Getting the User's Locale

```
package {
  import com.actionscriptbible.Example;
  import flash.events.*;
  import flash.external.ExternalInterface;
  import flash.globalization.LocaleID;
  public class ch41ex1 extends Example {
    public function ch41ex1() {
      trace("Browser's locale-------------------");
      trace(getLocaleFromBrowser());
      trace("HTTP Accept-Languages locales-------");
      (new GetLocalesFromRequest()).addEventListener(Event.COMPLETE, onReady);
    }
    protected function getLocaleFromBrowser():String {
      if (ExternalInterface.available) {
        return ExternalInterface.call("\
          function() {\
            var ret = null;\
            if (navigator) {\
              if (!ret) ret = navigator['language'];\
              if (!ret) ret = navigator['browserLanguage'];\
              if (!ret) ret = navigator['systemLanguage'];\
              if (!ret) ret = navigator['userLanguage'];\
            }\
            return ret;\
          }");
      }
      return null;
    }
    protected function onReady(event:Event):void {
      var response:GetLocalesFromRequest = GetLocalesFromRequest(event.target);
      for each (var pair:LocalePriorityPair in response.locales) {
        trace(pair.locale, "\t( preference =", pair.priority.toFixed(1), ")");
      }
      createLocaleID(response.bestLocale);
    }
```

continued

883

EXAMPLE 41-1 *(continued)*

```
    protected function createLocaleID(localeName:String):void {
      var locale:LocaleID = new LocaleID(localeName);
      trace("Locale information-------------------");
      trace("lang =", locale.getLanguage(), "\nregion =", locale.getRegion(),
        "\nscript =", locale.getScript(), "\nrtl? =", locale.isRightToLeft());
    }
  }
}
import flash.events.*;
import flash.net.*;
class GetLocalesFromRequest extends EventDispatcher {
  public var rawHeader:String;
  public var locales:Vector.<LocalePriorityPair>;
  public var bestLocale:String;
  public function GetLocalesFromRequest():void {
    var l:URLLoader = new URLLoader(new URLRequest(
      "http://actionscriptbible.com/files/accept-languages.php"));
    l.addEventListener(Event.COMPLETE, onLoad)
  }
  protected function onLoad(event:Event):void {
    rawHeader = URLLoader(event.target).data;
    locales = new Vector.<LocalePriorityPair>();
    var re:RegExp = /([\w\-\,]+);q=([\d\.]+)\,?/gs;
    if (re.test(rawHeader)) {
      re.lastIndex = 0;
      var match:Object;
      while (match = re.exec(rawHeader)) {
        var samePriorityLocales:Array = match[1].split(",");
        var priority:Number = parseFloat(match[2]);
        for each (var localeName:String in samePriorityLocales) {
          if (!bestLocale) bestLocale = localeName;
          locales.push(new LocalePriorityPair(localeName, priority));
        }
      }
    } else if (rawHeader.length > 0) {
        bestLocale = rawHeader;
        locales.push(new LocalePriorityPair(bestLocale, 1));
    }
    dispatchEvent(new Event(Event.COMPLETE));
  }
}
class LocalePriorityPair {
  public function LocalePriorityPair(l:String, p:Number) {
    locale = l; priority =p;
  }
  public var locale:String;
  public var priority:Number;
}
```

To get the browser's language from JavaScript, I've included a few properties of `navigator` that may be hiding the value I want (depending on what archaic browser you run it on), where one falls through to the next one if it fails to get any value. See Chapter 43 to learn more about interfacing with JavaScript.

Much of the code in the example is spent parsing the wacky `Accept-Language` headers. You can shift some of this parsing off to the server side, if you want, returning JSON or XML or URL-encoded variables or something like that.

Getting the Closest Match

Say you've painstakingly localized your application for six locales. The user specifies a list of three locales or languages that he's comfortable with, in a preferred order. You need to determine which locale to actually use based on the languages you support and the languages the user wants to see. The `LocaleID` class can help you determine that. Call its static method `determinePreferredLocales()`, passing a `Vector` of the user's locales (in decreasing order of his preference), and a `Vector` of the localized locales, and it solves this for you, returning a `Vector` of the locale(s) you should use.

For brevity's sake, in examples here, I'm either going to use a specific locale or the default one, rather than applying these locale-finding techniques.

Formatting Numbers

Format numbers and interpret numeric input in a locale-specific way with a `NumberFormatter` instance. Simply instantiate one, passing the constructor your locale code (just a `String`; you won't really use the `LocaleID` object), and you can convert text to numbers and numbers to text using these methods (demonstrated in Example 41-2):

- `formatNumber()` — Convert a `Number` into text in the object's locale.
- `formatInt()` — Convert an `int` into text in the object's locale.
- `formatUint()` — Convert a `uint` into text in the object's locale.
- `parseNumber()` — Convert a `String` containing only a single number in the object's locale into a `Number`.
- `parse()` — Convert a `String` containing a number in the object's locale, possibly along with non-number garbage, returning a `NumberParseResult`, which tells you the `startIndex` and `endIndex` of the number within the source string, as well as the `value` it found.

Additionally, there are a number of properties you can set that control the format of a number in the locale. I won't go into depth on these, because the point of this class is to know how numbers look in the supported locales so that you don't have to.

EXAMPLE 41-2   `http://actionscriptbible.com/ch41/ex2`

Localizing Numbers

```
package {
  import com.actionscriptbible.Example;
  import flash.globalization.NumberFormatter;
  public class ch41ex2 extends Example {
```

EXAMPLE 41-2 *(continued)*

```
    public function ch41ex2() {
      var usa:NumberFormatter = new NumberFormatter("en-US");
      var german:NumberFormatter = new NumberFormatter("de-DE");
      var russian:NumberFormatter = new NumberFormatter("ru-RU");
      var thai:NumberFormatter = new NumberFormatter("th-TH");
      var tamil:NumberFormatter = new NumberFormatter("ta-IN");
      var all:Vector.<NumberFormatter> =
        new <NumberFormatter>[usa, german, russian, thai, tamil];

      var n:Number = usa.parseNumber(" 5,619.02");
      for each (var formatter:NumberFormatter in all) {
        trace(formatter.requestedLocaleIDName,"\t",formatter.formatNumber(n));
      }
    }
  }
}
```

This example parses in a number, comma separator and all, the way it's written it in the States. (`parseFloat()` would choke on the comma, natch.) Then it spits it out in a variety of interesting locales. For example, in Germany, the separators are flipped and it's written 5.619,02, and in Russia, it's written 5 619,02. Tamil and Thai use non-Roman digits, but I won't force my poor editors to reproduce them here.

Formatting Dates

Dates and times can also be automatically converted by locale. As you found out in Chapter 7, "Numbers, Math, and Dates," dates can be pretty tricky, so the `DateTimeFormatter` is a godsend. Like other formatter classes, create a new `DateTimeFormatter` by passing in a locale code to the constructor. In addition, you tell the formatter how tersely it should represent the date and time. The options are constants of `DateTimeStyle`: NONE, SHORT, MEDIUM, LONG, and CUSTOM. Either pass these to the constructor:

```
//Short date, no time, aussie style
var australian:DateTimeFormatter = new DateTimeFormatter("en-AU",
  DateTimeStyle.SHORT, DateTimeStyle.NONE);
trace(australian.format(new Date(1981, 4, 12))); //12/05/1981
```

or set them with `setDateTimeStyles()`. You can retrieve them at any time using the analogous `getDateStyle()` and `getTimeStyle()` methods.

```
//Long date, no time, austria style
var austrian:DateTimeFormatter = new DateTimeFormatter("de-AT");
austrian.setDateTimeStyles(DateTimeStyle.LONG, DateTimeStyle.NONE);
trace(austrian.format(new Date(1981, 4, 12))); //Dienstag, 12. Mai 1981
```

Did you see how easy that was? In the words of someone who's had to painstakingly localize applications into languages he doesn't know: "booya."

Translate `Date` objects into localized text with the `format()` method. You can also use a `Date` object in UTC time by passing it to `formatUTC()` instead. Note that neither method messes with the time; for instance, the formatter won't try to convert from GMT-0500 Eastern time to GMT+0100 Austria time.

Beyond the flexible options given by `DateTimeStyle`, you can completely customize the formatting of a date by setting properties of the `DateTimeFormatter` instance. Again, I won't cover those here, because the whole point is that these classes are already populated with the correct formatting for the given locale.

Additionally, `DateTimeStyle` lets you grab those localized month and day-of-the-week names if you need them in other places. Use `getMonthNames()` and `getWeekdayNames()` for this. These take optional parameters that let you request the full name or a long or short abbreviation (Thursday, Thurs., Th., for instance). Example 41-3 shows several date formats in action.

Localizing Dates

```
package {
  import com.actionscriptbible.Example;
  import flash.globalization.*;
  public class ch41ex3 extends Example {
    public function ch41ex3() {
      var rogerDay:Date = new Date(1981, 4, 12);
      var all:Vector.<DateTimeFormatter> = new Vector.<DateTimeFormatter>();
      all.push(new DateTimeFormatter("ja-JP",        //Japanese/Japan
        DateTimeStyle.MEDIUM, DateTimeStyle.NONE));
      all.push(new DateTimeFormatter("he-IL",        //Hebrew/Israel
        DateTimeStyle.LONG, DateTimeStyle.NONE));
      all.push(new DateTimeFormatter("pt-PT",        //Portuguese/Portugal
        DateTimeStyle.LONG, DateTimeStyle.NONE));

      for each (var formatter:DateTimeFormatter in all) {
        trace(formatter.requestedLocaleIDName,"\t",formatter.format(rogerDay));
      }

      //I forget, what are the days of the week in Swedish again?
      trace((new DateTimeFormatter("sv")).getWeekdayNames().toString());
    }
  }
}
```

Formatting Currency

For currency, use the `CurrencyFormatter`, which has much the same structure as `NumberFormatter`, but it's endowed with the ability to format quantities in the local currency correctly. When you create a `CurrencyFormat` object, you pass its constructor a locale string like the other globalization classes, associating it with

- The locale — Which you can retrieve as `actualLocaleIDName`, and of course, you can access `requestedLocaleIDName` as well.

- The currency used in the locale — `currencyISOCode` contains the three-letter code (like USD) for the currency used in the locale, and `currencySymbol` contains the symbol (like $) used to write amounts in the local currency.

- Currency formatting rules — If digits are grouped, the pattern of the groups, the number of decimal places, the separator characters, and so on.

Together, these rules define not only the locale, but its currency and how it's written. This means that a `CurrencyFormat` is best used to represent monetary amounts in the region's own currency, not in foreign currencies.

Note

The three-letter currency codes, like CAD, EUR, HKD, GBP, and so on, are standardized by ISO 4217; find a reference at `http://bit.ly/ISO4217.` ∎

To localize a monetary amount, call `format()` on the `CurrencyFormat`, passing in a bare number, as Example 41-4 shows. The number is presumed to be a quantity in the locale's own currency and is treated as such. A second optional Boolean argument uses the currency symbol ($) instead of the currency code (USD) when set to `true`.

EXAMPLE 41-4 `http://actionscriptbible.com/ch41/ex4`

Localizing Currency

```
package {
  import com.actionscriptbible.Example;
  import flash.globalization.*;
  public class ch41ex4 extends Example {
    public function ch41ex4() {
      var local:CurrencyFormatter = new CurrencyFormatter(LocaleID.DEFAULT);
      trace(local.format(123456.78)); //USD123,456.78
      var usa:CurrencyFormatter = new CurrencyFormatter("en-US");
      trace(usa.format(123456.78, true)); //$123,456.78
      var jp:CurrencyFormatter = new CurrencyFormatter("ja-JP");
      trace(jp.format(123456.78, true)); //¥123,457
      var de:CurrencyFormatter = new CurrencyFormatter("de-DE");
      trace (de.format(123456.78, true)); //123.456,78 €
      var fr:CurrencyFormatter = new CurrencyFormatter("fr-FR");
      trace (fr.format(123456.78, true)); //123 456,78 €
```

```
      var localeID:LocaleID = new LocaleID(local.actualLocaleIDName);
      trace("Local currency in " + localeID.getRegion() + ": " +
        local.currencyISOCode + " (" + local.currencySymbol + ")");
    }
  }
}
```

Likewise, you can parse in a currency from localized text to a number, with the `parse()` method. When passed a `String` containing a currency in the local format, the method returns a `CurrencyParseResult` object with a `value` — the amount of money in the local currency — and a `currencyString` — the substring that contains the unit of currency as a currency code or symbol.

Localized String Comparison

Although the `String` class is perfectly adept at storing text in any language, some of its methods aren't intelligent enough to work in absolutely every case and every language. A particularly thorny problem is string ordering. Different languages that share the same alphabet sometimes order it differently. Languages treat accented characters differently. Sometimes letter pairs are considered one letter. And so on and so on. `String` comparison operators are incredibly stupid next to the complexity of sorting in almost any language.

Because every language has its own rules, localizing an ordered set of data, or data in a pivot table, requires writing sorting algorithms again and again for every supported locale. Moreover, the built-in comparisons don't even work for English most of the time, so you'd always have to write a custom sort function. No longer! Now you can use the `Collator` class.

After constructing the `Collator` class with the correct locale, use two of its methods to perform localized string comparison.

- `compare()` — Compares two `String` arguments and returns an `int` indicating the two arguments' relative order.
- `equals()` — Compares two `String` arguments and returns `true` if they are equal.

The `compare()` method can be used as an argument to `Array`'s `sort()` method, as Example 41-5 demonstrates.

EXAMPLE 41-5 `http://actionscriptbible.com/ch41/ex5`

Localized String Comparison

```
package {
  import com.actionscriptbible.Example;
  import flash.globalization.Collator;
  public class ch41ex5 extends Example {
    public function ch41ex5() {
      var words:Array;
      var a:String = "apples";
      var b:String = "Barnaby";
```

continued

EXAMPLE 41-5 *(continued)*

```
        var c:String = "calliope";
        var d:String = "detritus";
        var e:String = "éclair";
        var f:String = "firefight";

        words = [f, d, b, e, c, a];
        words = words.sort();
        trace(words); //Barnaby,apples,calliope,detritus,firefight,éclair

        words = [f, d, b, e, c, a];
        var usaCollator:Collator = new Collator("en");
        words = words.sort(usaCollator.compare);
        trace(words); //apples,Barnaby,calliope,detritus,éclair,firefight

        usaCollator.ignoreCase = true;
        usaCollator.ignoreDiacritics = true;
        trace(usaCollator.equals("Eclair", "éclair")); //true
      }
    }
  }
```

In this example, you use a naïve sort method on a list of English words. It's flummoxed by the mixed case of the words and an accented E for éclair. With use of an English-locale `Collator`, the sort works as expected.

Some properties of `Collator` can be modified to change the way a sort proceeds. In Example 41-5, the sort actually was case-sensitive, although capitals are placed correctly with their lowercase companions rather than before any lowercase letter. You can actually make `Collator` blind to not only letter case, but diacritics and more, as demonstrated at the end of the example, when "Eclair" and "éclair" are found to be equivalent.

These properties are affected by the mode the `Collator` is created in. Options can be set to favor sorting, as you did in the first half of the example, or pairwise matching, as you did in the second half of the example. Set the initial mode by passing `CollatorMode.SORTING` or `CollatorMode.MATCHING` as the second, optional argument to the constructor:

```
    var localSorter:Collator = new Collator(LocaleID.DEFAULT, CollatorMode.SORTING);
```

Sorting mode is selected by default if the argument is omitted.

The initial mode does nothing more than initially set properties of the `Collator` that affect comparison. These may all be set after construction time as well, and include

- `ignoreCase` — When `true`, a lowercase letter and its capital are considered equivalent.
- `ignoreCharacterWidth` — When `true`, half-width versions of Japanese and Chinese characters are treated identically to their full-width versions.
- `ignoreDiacritics` — When `true`, accents and diacritics are ignored.

- `ignoreKanaType` — When `true`, hiragana and katakana versions of the same syllable are treated identically.
- `ignoreSymbols` — When `true`, symbolic characters including whitespace, punctuation, currency, and math characters are ignored.
- `numericComparison` — When `true`, numbers in strings are treated numerically, rather than character-by-character. For example, using string sorting, "100" < "2" ("1" comes before "2"), but using numeric sorting, 2 < 100.

As in other globalization classes, these usually take on the right properties depending on their locale and the `CollatorMode` used (as you saw in Example 41-5, because the sort worked perfectly without modifying the `Collator`). Tweaking them individually can be useful, though — particularly `numericComparison`.

Localized Capitalization

In a few edge cases, `String`'s capitalization facilities also fall short. In general, `toUpperCase()` and `toLowerCase()` do an admirable job, handily changing the case of all kinds of accented Latin characters (covering languages from Afrikaans to Vietnamese), the Cyrillic alphabet, the Greek alphabet, and even Japan's full-width roman character sets. Where it falls short are a few cases where the capitalization rule is not so simple. For instance, German's lowercase ß should become an SS when capitalized, and Greek's lowercase sigma when in the word-end position, ς, should become a capital sigma Σ.

Of course, "just" screwing up one letter in the alphabet every time it appears is hardly acceptable. When you need to ensure that capitalization uses locale-specific rules, use the `StringTools` class, as shown in Example 41-6. After initializing it with a locale code, call its `toLowerCase()` and `toUpperCase()` methods instead of `String`'s.

EXAMPLE 41-6 `http://actionscriptbible.com/ch41/ex6`

Localized Capitalization Rules

```
package {
  import com.actionscriptbible.Example;
  import flash.globalization.StringTools;
  public class ch41ex6 extends Example {
    public function ch41ex6() {
      var odysseus:String = "Οδυσσεύς";
      var greekStringTools:StringTools = new StringTools("el-GR");
      trace(odysseus);
      trace(odysseus.toUpperCase());
      trace(greekStringTools.toUpperCase(odysseus));

      var biteme:String = "beißen mich";
      var germanStringTools:StringTools = new StringTools("de-DE");
      trace(biteme);
      trace(biteme.toUpperCase());
      trace(germanStringTools.toUpperCase(biteme));
    }
  }
}
```

Error Handling

All the preceding globalization classes handle errors the same way. Although error handling wasn't shown in these examples for brevity, you should check for errors in two places.

First, you should make sure that the locale the system provides is what you had in mind, or an acceptable alternative for, the locale you requested. Flash Player leans on the host operating system for localization support, so the supported locales may vary from system to system. There's no guarantee that when you request a locale, the system has an appropriate match. After constructing the globalization class, check its `actualLocaleIDName` against the `requestedLocaleIDName`. If an appropriate one can't be found, the system returns the default locale. Whether that's a fatal error for you or not depends on the requirements of your app, so be sure to check it out if it would be critical. You can also get the locales supported by a globalization class by calling its static `getAvailableLocaleIDNames()` method. This returns a `Vector` of locale names.

The classes covered in this section don't raise errors when performing localizations. Instead, they indicate either success or a wide variety of errors in their `lastOperationStatus` property. This way, they can automatically localize a whole interface, ignoring errors, and you don't need to stay up at night worrying that a strange translation in one language out of fifty supported languages causes an error that brings down the application. The localization will go on. That doesn't mean, of course, that you should ignore errors. Check the `lastOperationStatus` property frequently, or when the result of a localization operation is used elsewhere or is otherwise critical. The `LastOperationStatus` class enumerates a bunch of possible errors that may be found in that property.

Accessibility

Not everyone experiences your Flash application the same. Few people have perfect sight and hearing; some will use assistive software or hardware to make interacting with the computer easier, or possible at all. For users with limited hearing, you can include captions, transcripts, or picture-in-picture video of a sign language translator. Enabling applications for users with limited or no sight is more of a challenge. Assistive software that aids users with limited vision usually does so by picking out the interactive elements of a UI and enabling keyboard navigation between them, along with describing and reading graphics and text when possible.

Now, Flash Player gets a bad rap where accessibility is concerned. But the fact is, Flash Player enables truly deep integration with assistive software. But it's up to developers to make applications accessible. Not that this is the fault of developers, necessarily. Many Flash projects, if they're not highly visual by nature, are rushed to the finish line with no budget and schedule for QA, much less localization or accessibility concerns.

Blame game aside, you can do a lot to improve accessibility of your application merely by ensuring that UI elements are tab accessible with a sane tab ordering as discussed in Chapter 21, "Interactivity with Mouse and Keyboard Events."

Flash Player 10 and later support Microsoft Active Accessibility (MSAA) on Windows, as long as you are using the ActiveX version of the plug-in, and when the embedding parameter `wmode` is not `windowless` or `transparent`. This standard gives screen readers that support it the ability to intuit information about custom UIs in Flash Player, such as by assuming that noninput text inside a button is probably the button's label. To see more about what Flash Player 10 MSAA support can do for you, visit `http://bit.ly/fp10_accessibility`.

More importantly, you can provide assistive devices with meta-information about display objects with the `AccessibilityProperties` class. The class is trivial to use:

```
input.accessibilityProperties = new AccessibilityProperties();
input.accessibilityProperties.name = "Full name";
input.accessibilityProperties.description = "Your full name, first
then last.";
```

Caution

This `name` property is not the same as the `DisplayObject`'s `name` property, which screen readers ignore. ∎

The `accessibilityProperties` property can be set on any button, display object container, text field, or the application class. In other places, it is ignored. Simply adding `name` accessibility properties to buttons and `description` accessibility properties to groups of important controls does a world of good. With the `forceSimple` property of `AccessibilityProperties`, you can make the associated display object opaque to screen readers — they ignore its children.

```
group.accessibilityProperties = new AccessibilityProperties();
group.accessibilityProperties.description = "Three screen captures of \
the software in action.";
group.accessibilityProperties.forceSimple = true;
```

Similarly, there is an `accessibilityProperties.silent` property that hides the display object from screen readers when set to `true`.

Additionally, you can supply an alternate interface optimized for screen readers if one is active. You can detect screen reader software by using the static read-only property `Accessibility.active` in the `flash.accessibility` package.

But wait! Don't go wild setting thousands of accessibility properties around your application just yet. Flash Player's auto-labeling features may do a tolerable job for much of your app. If you can, get a free or open source screen reader, or a trial version, and find out where you should focus your efforts at clarifying the interface for assistive devices.

Color Correction

Flash Player 10 and later can use color correction when the host environment supports it. Color correction is used by software to ensure colors are reproduced consistently across different devices — in this case, displays, but color correction is also critical for accurate printing, scanning, and image acquisition. Different models of a display, and especially displays from different manufacturers, reproduce colors slightly differently. If the display's manufacturer is concerned at all about color accuracy, it provides you with a color profile for the display. The color profile tells software in great detail about how colors are produced by the device, something software alone could never find out, and usually using tightly controlled data (creating your own accurate color profile can't be done well without an optical color calibration device). Now that it has a color profile, your software is armed with knowledge about how your monitor translates the colors in memory to the exact colors that the hardware emits. With this knowledge, it can adjust the colors it asks to display, so that the ones that come out are exactly the intended colors in the first place. That's color correction.

893

Version

FP10. Flash Player color correction is available only in Flash Player 10 and later. ∎

Use color correction to ensure a consistent and accurate color experience between users with different displays. The effects of color correction may not be noticeable when the hardware is close to the default color profile used by Flash Player, but it's critical for image editing applications and precision video and image review.

Your operating system has color correction built in, but it may not be doing anything for you unless you have the correct color profile installed. Presume that you have one installed and working. The plug-in and ActiveX versions of Flash Player still have to rely on the host browser, which may or may not be applying color correction to its contents. In general, Flash Player tries to match the color correction settings of the browser. That way, if Flash content is integrated with non-Flash content, the adjusted colors don't clash.

Two properties on the `Stage` instance provide access to color correction.

- `colorCorrectionSupport` — Read-only; specifies whether Flash Player can use color correction and whether the browser is using it. If the monitor doesn't have an ICC color profile or Flash Player can't read the profile, the value is `ColorCorrectionSupport.UNSUPPORTED`. Otherwise, color correction can be used, and its default state is indicated by a value of `ColorCorrectionSupport.DEFAULT_ON` or `ColorCorrectionSupport .DEFAULT_OFF`.
- `colorCorrection` — Turns color correction on and off. Set to `ColorCorrection.ON`, `ColorCorrection.OFF`, or `ColorCorrection.DEFAULT` to inherit behavior from the host browser.

It's not really necessary to check the support first. If you want color correction to be on, just set it to on.

```
stage.colorCorrection = ColorCorrection.ON;
```

Turning on color correction uses more CPU time. You may not notice a difference when you toggle between them. If you want to see the effects, try creating a custom color profile with a software monitor calibration tool (your OS probably has one built in), and make sure its effects are exaggerated.

Color correction is especially important if your site was designed on all Macs or all PCs and if it's very dark or very bright. Macs and PCs traditionally display at different gamma levels, which are sure to lose details in the shadows and highlights. Enabling color correction can fix this.

Check out Example 36-12 (`http://actionscriptbible.com/ch36/ex12`) to see how you can use palette mapping to correct colors in pure ActionScript. You can use this approach in Flash Player versions earlier than 10.

Tip

Here's a massive power tip. Flash Player only color-corrects its color output. All external files and art are assumed to be in the sRGB color space. So when you're creating art assets for Flash Player, always work in, and save to, the sRGB color space. ∎

Summary

- Globalization, accessibility, and color correction all help ensure that everyone can enjoy your ActionScript application.
- Globalization, or internationalization, is the process of readying your application for use in multiple locales. Localization is the process of converting your application for a specific locale.
- A locale defines conventions for written text. It's at least a language and usually a combination of language and region, such as U.S. English (`en-US`).
- Flash Player 10.1 and later include a globalization package that makes localizing numbers, currency, dates, and times easy. It also aids locale-appropriate `String` comparisons and capitalization.
- It's not impossible to make Flash applications accessible, nor is it very hard.
- Use sane tab ordering, and when necessary, `AccessibilityProperties` objects attached to `InteractiveObjects`, to make your application sound more descriptive and organized to users without sight.
- The best way to test accessibility is to try your application in a screen reader.
- Color correction is used to reproduce colors accurately across display devices with different color characteristics.
- Flash Player 10 and later include software color correction, which typically does whatever the host browser does. Turn it on or off manually with the `stage.colorCorrection` property.

Deploying Flash on the Web

Programming your creation in ActionScript 3.0 is the first and most impor-
tant step. Any successful project must also be deployed correctly. In most
cases, Flash content is seen on the World Wide Web, within your web
browser. There are other possibilities for your content, such as a screensaver or a
kiosk, that typically require third-party tools to implement. AIR applications are
deployed for desktop operating systems, and Flash Professional CS5 and later can
package your application as an iPhone app. But this chapter covers the important
issues in preparing your Flash content for deployment on the web.

Embedding Flash in a Page

If you are developing Flash content for the web, at some point you have to
embed your SWF in an HTML page. This process is a stroll in the park, but
unfortunately the park is strewn with land mines.

Rendering Flash in a web page requires the Flash Player plug-in. Despite the fact
that browsers have been around for nearly two decades, there are still differences
in plug-in architectures and the HTML that should be used to embed them.
To wit, Internet Explorer uses ActiveX controls, and Mozilla browsers use the
NSPlugin architecture. Different tags have been used to embed content: the
`<embed>` tag and the `<object>` tag. Browsers even implement these tags in
slightly different ways. To complicate matters, writing valid [X]HTML is desirable,
and often required, but the `<embed>` tag is not valid in HTML 4 or XHTML 1,
although it *is* valid in HTML 5. To make things worse, the EOLAS patent suit
against Microsoft in 2005 (`http://en.wikipedia.org/wiki/Eolas`) forced
Internet Explorer to require a mouse click to activate plug-ins.

In a utopian future where everyone's browser fully supports HTML 5, you will
have to write just one standards-compliant `<object>` or `<embed>` tag to embed
Flash content. But at the present time, you have to choose a method of embed-
ding SWFs that will provide your content to people with all kinds of browsers,

without using invalid markup, that gets around the click-to-activate issue. It needs to be able to display alternative content if the user does not have Flash, and it must be able to detect the version of Flash Player that you are targeting.

Caution

Without intervention, Flash Player tries to play SWFs published for any version of Flash Player. Flash Player 9 tries to display SWFs published for Flash Player 10. Backward compatibility is important to Flash Player: playing a SWF made for an older version of Flash Player should never be a problem. However, forward compatibility does not really exist. If you have any ActionScript in your Flash content, you should not allow older Flash Players to play your content. Accordingly, you should publish your SWFs to the lowest version of Flash Player that supports all the features you use. If I haven't made this clear yet, *all ActionScript 3.0 programs require Flash Player 9 or later*. ∎

Fortunately, several tools have stepped up to fulfill these goals. The culmination of these is SWFObject (`http://swfobject.googlecode.com`), which is the marriage of two well-tested SWF embedding toolkits. It is the de facto standard, it is tested and updated regularly, and I recommend using it.

Flash CS4 Professional and Flex Builder 3 shipped with Adobe's ActiveContent JavaScript embedding solution, another embedding toolkit. Flash Professional CS5, and Flash Builder 4, on the other hand, use SWFObject.

Alternatively, there are some HTML-only solutions that work with one or two exceptions to embed content cross-browser. The *nested object* solution and the *Flash satay* solution are succinct and standards compliant, but they have their limitations. SWFObject in static publishing mode uses the double-object method that works without JavaScript but builds on it when JavaScript is available to correct some browsers' bugs.

As an added reason to use SWFObject, the web development community has produced some excellent extensions to SWFObject. For example, SWFAddress (`http://asual.com/swfaddress/`) makes it simple to provide browser history integration and deep linking. SWFMacMouseWheel (`http://blog.pixelbreaker.com/flash/swfmacmousewheel/`) uses JavaScript to add support for the mouse wheel on the Mac platform. SWFFit (`http://swffit.millermedeiros.com/`) forces the browser to display scroll bars when a full-page SWF is resized below certain minimum dimensions. All these extensions greatly improve the usability of Flash applications, and I recommend that you use them when appropriate.

Embedding Flash Using SWFObject

The main purpose of SWFObject is to be able to show your Flash content in any browser, but only if you have the appropriate version of Flash Player, showing you alternative content if you do not. Although the ultimate destination for information about SWFObject is its home page at `http://swfobject.googlecode.com/`, a quick introduction of the SWFObject syntax here provides context for the examples in the rest of the chapter.

SWFObject works in two modes: static publishing and dynamic publishing. Using static publishing, HTML is written that includes the tags necessary to embed the SWF. This way, if JavaScript is not available, the Flash content may still be run, although with some browser bugs (that are fixed if JavaScript is available and you use SWFObject). In dynamic publishing mode, the proper embedding

code is written into the page contents by JavaScript when the page renders. Both have their advantages, which are outlined in the SWFObject documentation (`http://code.google.com/p/swfobject/wiki/documentation`). I will cover the dynamic publishing method. Use whichever suits you.

A simple HTML page that embeds Flash content using SWFObject might be written as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Demo</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <script type="text/javascript" src="swfobject.js"></script>
  <script type="text/javascript">
    swfobject.embedSWF("demo.swf", "alternate-content", "500", "500", "9.0.0");
  </script>
</head>
<body>
  <div id="alternate-content">
    Please <a href="http://www.adobe.com/go/getflashplayer">Get Flash Player</a>
    to see this content.
  </div>
</body>
</html>
```

When using SWFObject with dynamic publishing, you need to include the following three elements in your HTML page:

- A `<script>` tag to include the SWFObject code. Download `swfobject.js` from the SWFObject web page, upload it to your own site, and include it as in the preceding example.

- A `<div>` containing an HTML alternative to your Flash content, with an `id`. Think of this like an `alt` tag on an image. The HTML content will not be displayed if the user is capable of displaying the Flash content. It will appear to any search engine.

- A `<script>` block that calls the dynamic publishing method `swfobject.embedSWF()`.

SWFObject replaces the content of the `<div>` named in `embedSWF()` with the HTML necessary to embed the Flash content as you specified using the SWFObject object. So you have a `<div>` named "alternate-content" in which `demo.swf` is embedded, replacing its HTML content.

The most important — and required — parameters to the `embedSWF()` method are as follows:

- The URL to the SWF that will be embedded.

- The `id` of the HTML element to replace with Flash content.

- The width that Flash Player should take up (in any measurements understood by CSS).

- The height that Flash Player should take up.

- The minimum version of Flash Player that can correctly run the content. Lower versions will see the alternate content, or automatically upgrade if you enable this option.

You can see the full list of parameters at SWFObject's documentation page.

When creating an embed page, you may find that it's easiest to use the SWFObject generator (`http://code.google.com/p/swfobject/wiki/generator`) to create your HTML, or an appropriate template for it. It allows you to customize your embed interactively.

Enabling Flash Player Options

Regardless of which embedding technique you use, the tags used to embed Flash content can convey a variety of options to Flash Player. These are fully documented in the Flash Professional help. See `http://bit.ly/embed-parameters`.

When an `<embed>` tag is used, these options are set as attributes of the embed tag. When an `<object>` tag is used, the options are encoded as a sequence of child nodes of the form `<param name="" value=""/>`. Use this approach when writing HTML by hand. For example, the following code sets the quality of a Flash movie using an `<embed>` tag in HTML:

```
<embed src="foo.swf" quality="high"></embed>
```

To set the same property using an `<object>` tag, use this:

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000">
    <param name="movie" value="foo.swf"/>
    <param name="quality" value="high"/>
</object>
```

SWFObject lets you set embedding options in the `embedSWF()` method. Pass a JavaScript `Object` as the `params` argument to the function. This block of JavaScript replaces the div with an `id` of `flashcontent` with the Flash movie, and it sets the quality:

```
<script type="text/javascript">
  params = {quality: "high"};
  swfobject.embedSWF("demo.swf", "alternative-content",
    "500", "500", "9.0.0", false, {}, params);
</script>
```

When using static publishing with SWFObject, set the properties on both `<object>` tags by hand (or using the generator), as described earlier.

Although the documentation covers all embedding parameters, I will describe some particularly useful ones next.

Transparent Flash

By default, the Flash Player plug-in renders your Flash content with an opaque background and without being composited against other elements in the browser. By setting the `wmode` parameter of the embedding code to `transparent`, you can make Flash content render without a background, composited with all other HTML elements in a page. This can draw SWFs with irregular borders, expand Flash content by bleeding into an otherwise transparent region, and convincingly layer Flash content with HTML content.

Caution

Flash content embedded as transparent takes up more system resources and performs more slowly than its opaque counterpart. Use transparent Flash sparingly! ■

Hardware Acceleration

Different values for the `wmode` embed parameter can enable some of Flash Player's hardware acceleration, controlling how Flash Player pushes graphics to your screen for, in many cases, enhanced performance. The hardware acceleration comes in two flavors, each with a `wmode` parameter.

- `direct` — Uses direct mode rendering if possible, bypassing the browser's rendering code and drawing directly to whatever display API your OS exposes (DirectDraw, DirectX, or OpenGL). This mode can improve performance in games, video players, and other applications that redraw large areas of the screen.
- `gpu` — Uses GPU mode rendering using the OS's display API (DirectX or OpenGL) to composite frames after drawing, much like Mac OS X and Vista Aero composite windows on the graphics card. This doesn't speed up drawing, only compositing, and not in all cases. Also, requesting GPU mode using the `wmode` parameter doesn't guarantee it will be active. Hardware and software requirements must be met. Use the `Stage` instance's `wmodeGPU` parameter to determine whether GPU mode is being used.

Hardware-accelerated Flash requires a modern graphics card and operating system. Acceleration won't improve performance in all cases. For a brief discussion of the GPU-accelerated rendering modes, read Tinic Uro's post at `http://bit.ly/kaourantin-gpu-acceleration`.

Version

FP 10. Hardware-accelerated rendering is only available in Flash Player 10 and up. ■

Full-Screen Flash

You can use ActionScript to put the Flash Player in full-screen mode, overtaking the user's entire screen with your content. You can even enable this behavior in a browser. This allows a Flash video player, for instance, to display in its normal size inside the browser window or to fill the user's entire display. I covered this in Chapter 14, "Visual Programming with the Display List."

Version

FP9. Full-screen mode is only available in Flash Player 9.0.28 and later. ■

For the Flash Player plug-in to enable full-screen mode, the `allowFullScreen` embed parameter must be set to `true`, as in the following:

```
params = {allowFullScreen: "true"};
swfobject.embedSWF("demo.swf", "alternative-content",
  "500", "500", "9.0.0", false, {}, params);
```

Because this feature requires Flash Player 9.0.28 or later, be sure to configure your Flash embed to require this version or later to ensure full-screen capability.

901

Passing Variables to a SWF

You can pass values from HTML into embedded Flash content. These variables are often inserted into the HTML by scripts or passed in via the URL query string. You can use this technique to enable one SWF to serve in many situations, to avoid adding a second network access to retrieve some value that is available at the time the HTML page is served, or to ensure that a value is set before the first frame of the SWF plays. For example, sIFR (http://www.mikeindustries.com/sifr/) uses one SWF file (per font) to display any given text in a specific typeface by passing the text to be rendered into the Flash content at embed time.

Passing values into embedded SWFs, in fact, is also achieved by setting an option on the embedding tags. The option name is `flashVars`, and its value is a URL-encoded set of name/value pairs. By using URL encoding, you can represent multiple variable names and values in one string so that you can use that string as the value of a single Flash embedding option. The following example illustrates this phenomenon. If you have the following variables

```
var name = "Roger";
var age = 28;
var eyes = "brown";
```

you can send these to an embedded SWF by setting the `flashVars` option:

```
<param name="flashVars" value="name=Roger&age=28&eyes=brown"/>
```

Because of the URL encoding, all variables passed to an embedded SWF are encoded as strings. You can include as many variables as you want, but the whole URL-encoded string must be under 64KB or browsers may not accept it.

SWFObject lets you send variables into embedded SWFs without dealing with URL encoding. Just pass in a JavaScript `Object` containing all your variables to the `flashvars` argument of the `embedSWF()` function.

```
flashvars = {name: "Roger", age: 28, eyes: "brown"};
swfobject.embedSWF("demo.swf", "alternative-content",
  "500", "500", "9.0.0", false, flashvars);
```

To access these variables in the embedded SWF, access the `LoaderInfo` object of the root class of that SWF. This is the Application class if you're using Flash Builder, or it's the class set to the Document Class if you're using Flash Professional. The `parameters` property of the `LoaderInfo` object is an `Object` in which you will find all the variables passed in at embed time. Example 42-1, when used as the root class, traces out all the variables passed in by the embed.

EXAMPLE 42-1   http://actionscriptbible.com/ch42/ex1

Accessing Variables Passed to the SWF

```
package {
  import com.actionscriptbible.Example;
  public class ch42ex1 extends Example {
    public function ch42ex1() {
      var flashVars:Object = this.loaderInfo.parameters;
```

```
      trace("flashvars ----------");
      for (var variable:String in flashVars) {
        trace(variable + ": " + flashVars[variable]);
      }
    }
  }
}
```

Automatically Upgrading Flash Player

To ensure that users see your content with an appropriate version of Flash Player, you must ask those with older versions to upgrade. However, you want to make this as easy as possible. Instead of redirecting users to Adobe's Flash Player install page, you can actually trigger Flash Player to upgrade itself without navigating away from the page using a technique called Express Install. Express Install will display a dialog box on the page where your SWF would normally display, asking only that you confirm the upgrade. Figure 42-1 shows what the user will see in place of the Flash content.

FIGURE 42-1

The Express Install interface

To follow an Express Install process, you must first determine that the user's Flash Player needs to be upgraded. Then you typically pass off installation to an alternate SWF. This SWF, to be able to upgrade the oldest possible Flash Player version, should be compiled for the earliest version of Flash Player that supports Express Install, Flash Player 6.0.65. This SWF then loads an upgrader SWF from Adobe: `http://fpdownload.adobe.com/pub/flashplayer/update/current/swf/autoUpdater.swf`. Adobe's SWF displays a dialog box to the user requesting permission to upgrade the Flash Player, and if permission is granted, attempts to install the latest Flash Player. When it is complete, Adobe's updater SWF signals a status back to your SWF, letting you know if the update was successful, failed, or was denied by the user. This description of the Express Install process is mostly for educational purposes. SWFObject comes with a template FLA to customize your Express Install experience, and in most cases, using the default Express Install SWF provided by SWFObject suffices. Once you have an Express Install SWF, pass its URL as the `expressInstallSwfUrl` parameter to `embedSWF()`. The following snippet uses the provided upgrade SWF called `expressInstall.swf`.

```
    swfobject.embedSWF("demo.swf", "alternative-content",
      "500", "500", "9.0.0", "expressInstall.swf");
```

903

Flash Builder can generate and publish an Express Install SWF for your project if you choose to use its HTML embedding templates. Check the Use Express Install option in a project's ActionScript Compiler properties panel.

To use Express Install, the user must have at minimum Flash Player 6.0.65, and the SWF it is running in must be at least 214 pixels wide by 137 pixels high. The user must also be able to install software on her own computer, which is often not the case for corporate or shared computers.

Summary

- There are many ways to embed Flash content on the web.
- There are scripted solutions and tag-only solutions.
- SWFObject is the de facto standard.
- Flash content can be embedded with many options.
- Variables can be passed into embedded SWFs using `flashVars`.
- The Flash Player plug-in can attempt to update itself using Express Install.

Interfacing with JavaScript

I n this chapter, I'll look at how you can communicate from Flash Player to JavaScript and vice versa. When you deploy Flash content in a web browser, you may want to be able to communicate with the container HTML page. You may want to call JavaScript functions from the SWF, and you may want JavaScript functions to be able to call functions within the SWF. Tightly integrating Flash Player and its container page yields a hybrid page that has the potential to be more accessible, quicker loading, or more standards compliant. You can play to the strengths of each technology, using small instances of Flash Player for graphically rich, multimedia, and highly interactive content, and using HTML for long stretches of fully accessible copy, dynamic CSS-directed layout, and CMS-driven content. With the glue described in this chapter, you can ensure these two pieces are tied together.

Using ExternalInterface

You can use the `ExternalInterface` class in the `flash.external` package to call JavaScript functions from Flash and call ActionScript functions from JavaScript. `ExternalInterface` is officially supported on the following browsers:

- Firefox 1.0 and above for Windows and Mac, 1.5.0.7 and above for Linux
- Safari 1.3 and above for Mac
- Internet Explorer 5.0 and above for Windows

Discontinued browsers Netscape and Mozilla are also officially supported. In addition, I have had no problems using `ExternalInterface` with Opera or Chrome. You can reasonably expect it to be supported by your users.

To ensure `ExternalInterface` is supported before using it, use its static `available` property.

Calling JavaScript Functions from Flash

You can use `ExternalInterface` to call a JavaScript function from within your SWF. The JavaScript function can then return a value to Flash Player. Simply call the static `ExternalInterface.call()` method. The `call()` method requires at least one parameter — the name of the JavaScript function to call. You can also pass additional parameters, each of which is passed to the JavaScript function as a function argument. The following code calls the JavaScript `alert()` function with a parameter of `hello`:

```
ExternalInterface.call("alert", "hello");
```

If the JavaScript function returns a value, that value will be returned by `call()`. Furthermore, JavaScript is executed synchronously: `call()` waits for the function it proxies to complete before evaluating, and Flash Player only proceeds to the next ActionScript statement once the result of the JavaScript function is computed. So you can treat `call()` as a transparent proxy for your JavaScript functions. The following example code assigns the return value from a JavaScript function `getStringValue()` to a variable:

```
var value:String = ExternalInterface.call("getStringValue");
```

Of course, the JavaScript function you call must be available in the same context as the Flash Player embed. Usually, these are defined in `<script>` blocks or loaded by the browser from JS files. If you use JavaScript to embed your SWF, you can ensure that the page has finished loading before running SWF content, preventing the unlikely condition that your `ExternalInterface` call might execute before the JavaScript function it calls is defined. Should you call an undefined JavaScript method, `undefined` is returned.

If your Flash application needs to execute JavaScript that isn't defined in the page, and for some reason you can't add this JavaScript code, you can use the following trick. Instead of a function reference, pass `call()` the function itself. In Example 43-1, you use an anonymous function to execute JavaScript that's not defined in a `<script>` tag.

EXAMPLE 43-1   `http://actionscriptbible.com/ch43/ex1`

Executing JavaScript with an Anonymous Function

```
package {
  import com.actionscriptbible.Example;

  import flash.external.ExternalInterface;

  public class ch43ex1 extends Example {
    public function ch43ex1() {
      if (ExternalInterface.available) {
        trace(ExternalInterface.call("function(){return navigator.userAgent}"));
      }
    }
  }
}
```

The example traces the browser's identity from within Flash Player, using an extremely simple function you write inside your ActionScript file (as a `String`). Using a short function like this is innocuous enough, but you should steer clear of passing long functions.

Note

You may be wondering how ActionScript and JavaScript can be compatible enough to pass data back and forth. ActionScript — being an ECMAScript language — shares all the basic data types of JavaScript, like `String`, `Array`, and `Object`. On the other hand, ActionScript defines many more data types that don't exist in JavaScript. Because you're calling JavaScript code, you have no reason to use these data types as function arguments, nor could a JavaScript function return them. ∎

Calling ActionScript Functions from JavaScript

You can also call ActionScript functions from JavaScript. To do so you must register the function in ActionScript so that it is accessible from JavaScript and then call the function from JavaScript via the Flash embed object.

You can expose an ActionScript function to JavaScript with the static `ExternalInterface` `.addCallback()` method. The method requires two parameters: the name of the function as you want to call it from JavaScript and the reference to the function or method that you want to register. The following example registers a function called `resume()` so that you can call it from JavaScript as `resumeFlashGame()`:

```
ExternalInterface.addCallback("resumeFlashGame", resume);
```

Once a function is registered, you can call it from within JavaScript. From JavaScript, you call the function as a method of the Flash object by referencing the ID of the plug-in or ActiveX object. The standard way to do this is with the JavaScript `getElementById()` function, defined in HTML DOM Level 2. This is compatible with all modern browsers. If your Flash content was embedded with an ID of `"flashObjectID"`, you would retrieve a reference to the plug-in object like so:

```
document.getElementById("flashObjectID");
```

And you could call the registered ActionScript method from JavaScript with this code:

```
document.getElementById("flashObjectID").resumeFlashGame();
```

or:

```
window.flashObjectID.resumeFlashGame();
```

You can pass parameters from JavaScript to the function (again limited by the data types available in JavaScript). Likewise, you can use the return value of the ActionScript function in JavaScript.

You should be able to determine the ID of the Flash Player embed element in JavaScript. However, should you need it, `ExternalInterface` can report the ID of the Flash Player element it's running inside. Use the static property `ExternalInterface.objectID`.

JavaScript Interaction and Flash Player Security

To interface with scripts in the containing web page, the `AllowScriptAccess` embed parameter must be set to the proper value. These values include

- `always` — Flash Player is allowed to communicate with the page, even if it is hosted in mismatching domains.
- `sameDomain` — SWFs may communicate with HTML from the same domain only. This is the default value.
- `never` — This prohibits the embedded SWF from communicating to JavaScript in the page.

See more about this embedding tag at `http://bit.ly/allow-script-access`. Likewise, to return values to ActionScript and register callbacks, you must allow access to the SWF from the domain the HTML is hosted in using `Security.allowDomain()`.

If the SWF is not permitted by these rules to communicate with the HTML page, `SecurityErrors` are raised.

Making a Hybrid Application with ExternalInterface

You've had a chance to read about the theory of `ExternalInterface`. Now you'll use the theory to put together a simple demonstration application. Example 43-2 uses a Flash movie with a rotating rectangle, a start/stop button, and a text field. The Flash movie is placed within an HTML page with a text input and a button. The Flash movie requests that data and displays it in the text field. When the user clicks the start/stop button in the Flash movie, it pauses and resumes the rotation of the rectangle, and it sends a message to the HTML text input to display the current status of the rectangle. The HTML button makes a new random color and sends it to the Flash movie. The Flash movie then applies that color to the rectangle.

EXAMPLE 43-2   `http://actionscriptbible.com/ch43/ex2`

Exposing ActionScript Functions to JavaScript

```
package {
  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.external.ExternalInterface;
  import flash.utils.Timer;

  public class ch43ex2 extends Sprite {
    private var rectangle:Shape;
    private var timer:Timer;
    public function ch43ex2() {
      rectangle = new Shape();
      rectangle.graphics.lineStyle(5, 0, 1);
      rectangle.graphics.drawRect(0, 10, 100, 100);
```

```
        addChild(rectangle);
        timer = new Timer(50);
        timer.addEventListener(TimerEvent.TIMER, timerHandler);
        togglePlayback();
        if (ExternalInterface.available) {
          ExternalInterface.addCallback("togglePlayback", togglePlayback);
        }
      }
    public function togglePlayback():void {
        if (timer.running) {
          timer.stop();
        } else {
          timer.start();
        }
      }
    private function timerHandler(event:TimerEvent):void {
        if (stage) {
          rectangle.x += 5;
          if (rectangle.x > stage.stageWidth) {
            rectangle.x = 0;
          }
        }
      }
    }
  }
}
```

Publish the example as `ch43ex2.swf`. Use `SWFObject` to embed the Flash content in an HTML page. You can learn more about using SWFObject in Chapter 42, "Deploying Flash on the Web." If you don't already have the SWFObject JavaScript file, download it from `http://swfobject.googlecode.com/` and place the JS file in the same directory as the SWF file. In the same directory as the SWF and JS files, add a new HTML document called `ch43ex2.html` with the following contents:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Example ExternalInterface</title>
<script type="text/javascript" src="swfobject.js"></script>
<style type="text/css">
  body {
    background-color: #eeeeee;
  }
  #buttons {
    width: 300px;
    overflow: auto;
  }
  #flashcontent {
    float: left;
```

```
      margin: 15px 20px;
    }
</style>
</head>
<body>
  <div id="flashcontent">
    <strong>You need to upgrade your Flash Player</strong>
  </div>

  <script type="text/javascript">
    swfobject.embedSWF("ch43ex2.swf", "flashcontent", "550", "400", "9");
  </script>
  <div id="buttons">
    <input type="button"
      onClick="document.getElementById('flashcontent').togglePlayback()"
      value="Toggle Playback" />
  </div>

</body>
</html>
```

Because I can't modify the containing HTML on wonderfl, you'll notice that these examples are hosted on `http://actionscriptbible.com/` instead.

Summary

- You can use `ExternalInterface` to create integrated applications in which Flash and JavaScript communicate synchronously.
- Use `ExternalInterface.call()` to call JavaScript functions from Flash.
- Use `ExternalInterface.addCallback()` to register an ActionScript function or method so that it can be called from JavaScript.

Local Connections between Flash Applications

U sing the `LocalConnection` class, you can enable multiple Flash applications running on the same computer to communicate with each other. This is interprocess communication (IPC) for the Flash platform. I covered client-server communication in Chapter 28, "Communicating with Remote Services": here I introduce client-client communication. Local connections work without a network connection of any kind, directly between ActionScript running in both applications.

Local Connections and Their Uses

Local connections enable communication between one or more Flash platform applications on the same computer without additional tooling. Just like most of the client-server techniques used in Chapter 28, by "communication" I specifically mean "method invocation." After you open a local connection from a sending application A to a receiving application B, application A can call methods, optionally passing arguments, to application B. This connection is one way in that A can only call methods on B. However, you can program a simple handshaking protocol that makes B establish a parallel connection back to A so that one connection exists for each direction, simulating bidirectional communication. Furthermore, the one-directional communication extends to return values: A can call methods on B, but any return values from B methods are lost; they are not sent back to A. The method invocations are asynchronous, but A can listen for an event indicating whether the message was successfully sent to B.

Local connections are convenient because they don't impose additional requirements: any two Flash platform applications on the same computer can communicate (assuming they don't violate the security policy). You don't need to open a network connection or even have a network adapter in the computer. You don't need to call out to the application's container, JavaScript, COM, or anything. The Flash runtime handles all of it. I've been implying another powerful advantage of local connections by saying "Flash platform" and "Flash runtime" rather than "Flash Player." Local connections work between different Flash runtimes, as well.

So you can communicate between ActionScript running in an AIR application and ActionScript running in Flash Player in the browser.

Another important property of local connections is that they work in all versions of Flash Player, including those that run AVM1 (pre-ActionScript 3.0; Flash Player 8 and below). Recall that AVM1 and AVM2 are completely separate virtual machines that run isolated from each other, so when you load in a SWF that was published for Flash Player 8 or earlier, your Action-Script 3.0 code has no access to the ActionScript 1.0/2.0 code in the loaded SWF. But both can use local connections! So `LocalConnection` can act as the bridge between SWFs running in AVM1 and AVM2; it can interface ActionScript 1.0 and 2.0 code with ActionScript 3.0 code.

There are several good but specific applications for local connections:

- Interfacing with a loaded SWF that uses AVM1
- Ensuring that only one instance of an application is running at once
- Using a web page to check if an AIR application is running, and communicating with it
- Linking Flash elements in a hybrid Flash/HTML site
- Linking multiple advertisement units in a single web page, making them interact cleverly

These are just some ideas. Any time you have to make Flash applications interact on the same computer, `LocalConnection` might be able to help.

There are obvious security concerns with letting Flash applications talk to each other; Flash Player or AIR can deny any local connection that doesn't satisfy its security policy. Security and local connections are discussed later in this chapter.

Implementing a Local Connection

There are two sides to any connection: the sending application and the receiving application. Let's see how to code each side.

The Sending Application

Creating and sending data over a `LocalConnection` is a cinch:

1. Create a `LocalConnection` object.
2. Send methods to the receiving application.
3. Optionally, listen for the status of those method invocations.

To create a new `LocalConnection` object, just use its constructor, which takes no arguments:

```
var sendingConnection:LocalConnection = new LocalConnection();
```

Sending Methods to a Receiving Application

With client-server communication, you pick a specific server to communicate with by its URL or domain/IP and port. But local connections can be made to any listening Flash application running

on the computer, regardless of where the SWF is hosted or what Flash runtime it's using. So it's up to you to specify who to talk to. You do this by agreeing to a channel to communicate on.

Say you're going on a trip into the deep woods with a hiker friend of yours. You agree ahead of time with your friend on a radio frequency to communicate on, and you both go your separate ways. You don't know what other hikers are out there in the woods listening to your radio, and without broadcasting, you don't even know if your friend is listening. So you turn on your radio and broadcast over the predetermined frequency. Your friend hears your signal and can listen to you. That's just how the sending application finds the listening application. There's also a chance of a rather unlikely problem here: a different hiker might be using the same frequency and might reply instead of your friend. In the same way, it's best to come up with a channel name for your `LocalConnection` that's unique.

Unlike client-server applications, where you should ensure a connection is established before sending data over it, you can just broadcast over your channel at will with local connections. Just like in the woods, you broadcast on your frequency without knowing who is listening; it's up to your friend to turn on his radio and tune it into that frequency.

So let's invoke a method on listening applications. To do so, use the `send()` method. You'll have to pass this method the channel name, the name of the function you'd like to invoke on the listening application, and optionally any number of arguments.

```
sendingConnection.send("mouseChannel", "mouseMoved", 234, 107);
```

In this line you're attempting to tell an application listening to channel `mouseChannel` to call `mouseMoved(234, 107)`, to set a new mouse position perhaps.

Note
A `LocalConnection` is limited to sending 40KB of data at a time; if you pass more to `send()`, it raises an `ArgumentError`. ∎

Listening for the Status of Local Method Invocations

The `LocalConnection` object broadcasts a `StatusEvent.STATUS` event for every method invocation you send to the receiving application. This event can indicate either success or failure of the call, shown by the `level` property of the event object. However, this is somewhat misleading. The status event doesn't indicate whether the code on the receiving application executed without error. It merely indicates whether the invocation message was received by a receiving `LocalConnection` listening on that channel.

In the radio example, the status event indicates whether someone heard your broadcast. It does not indicate whether the listener acted on your message. You can't really know that, because your friend can only listen and you can only send.

Listen to `StatusEvent.STATUS` to make sure your method invocation was received. If the `level` property of the event object is `"status"`, the message was received. If the level property is `"error"`, there was no receiving application listening on that channel.

Caution
Status events with a `level` of `"error"` are considered asynchronous errors by Flash Player, so they present an error dialog in debugging versions of Flash Player. Make sure to subscribe to status events so that errors are captured. ∎

Before sending the first invocation, you might add a status listener to the sending application:

```
sendingConnection.addEventListener(StatusEvent.STATUS, onSendStatus);

protected function onSendStatus(event:StatusEvent):void {
  if (event.level == "error") {
    trace("ERROR: local connection failed to send the message.");
  }
}
```

And that's all it takes to invoke methods on local applications. Note that you didn't need to connect or disconnect. This is not necessary because sending `LocalConnections` are broadcast only; they are stateless.

The Receiving Application

Listening to method calls from another local application using `LocalConnection` is also relatively easy, although the process is not totally analogous to the sender application:

1. Create a `LocalConnection` object and, optionally, assign it a delegate that contains functions that may be invoked by another local application.

2. Open the `LocalConnection` to messages by listening to a particular channel.

3. Methods are called by the `LocalConnection` as received.

4. Disconnect from the channel when you are finished receiving method invocations.

Both sending and receiving applications use the `LocalConnection` class. To implement the receiving application, first create a new `LocalConnection`.

```
var listeningConnection:LocalConnection = new LocalConnection();
```

Assigning a Delegate

When the sending application calls functions by name, the receiving application has to find those functions. By default, it looks to the receiving `LocalConnection` instance for those functions. For instance, if `listeningConnection` receives a method invocation for a function called `mouseMoved`, it attempts to call the function `listeningConnection.mouseMoved()`. There's just one problem with this: `LocalConnection` is not a dynamic class, so you can't add a `mouseMoved` function to it.

Not to worry, though, because `LocalConnection` lets you specify a delegate object. The delegate is the object that the received function calls are executed on. Set this property of a `LocalConnection` with its `client` property. For example, to have functions in the same context called, just set the delegate to `this`:

```
listeningConnection.client = this;
```

Or perhaps you could limit the local connection to certain functions by creating a delegate object with function references or inline functions. Assuming you have functions `mouseMoved()` and `mouseClicked()` defined in scope, you can create a delegate object limited to these two methods like so:

```
listeningConnection.client = {
  mouseMoved: mouseMoved,
  mouseClicked: mouseClicked
};
```

By creating your own delegate object like this, you can not only limit the functions that can be called, but map them from different names and pull them in from various different classes. You can also define the functions inline with this approach.

Listening to a Named Channel

Before you can receive any messages, you have to tune in to the proper channel. Simply call `connect()` with the name of the channel:

```
listeningConnection.connect("mouseChannel");
```

The channel name has some impact on the security policy, which I'll cover in the next section.

That's all! You have a sending connection and a receiving connection. Methods can now be invoked across these two applications on the same computer. The functions called by `send()` on the sending application execute the corresponding functions on the `client` object of the connected receiving application.

Closing the Connection

When you want to stop receiving from the local connection, just call its `close()` method:

```
listeningConnection.close();
```

Local Connections and the Security Policy

By default, local connections work between Flash applications hosted on the same domain (or between SWFs on your local filesystem). That is, if a sending application is being run from `actionscriptbible.com`, it defaults to broadcasting to other applications on `actionscriptbible.com`. With just a few changes, however, you can configure applications to send and receive messages across domains so that, for example, an application loaded from `actionscriptbible.com` can communicate with an application running on the same machine but loaded from `wiley.com`.

The channel name you choose has a big impact on cross-domain local connections. Thus far, you've simply chosen a channel name. Behind the curtains, however Flash Player has actually been modifying the channel names. When a SWF hosted on `www.actionscriptbible.com` connects to a channel `mychannel`, Flash Player expands this channel name into the following:

```
actionscriptbible.com:mychannel
```

Note
It is actually the superdomain that is added to the channel name, not the exact domain. You can see here that the `www.` was stripped from the domain. ■

Thus, if the sending application is on `senddomain.com` and the receiving application is on `receivedomain.com`, the sending application sends to channel `mychannel`, and the receiving application connects to channel `mychannel`. Although the channels seem to be the same, they will actually be mismatched after Flash Player expands them:

```
receivedomain.com:mychannel
senddomain.com:mychannel
```

Local connections are unlike most other security policy concerns that operate through `crossdomain.xml` files. You have several options to get around this restriction.

Specifying the Domain in the Sender

If you know the domain of the receiving application, you can specify this in the channel name when calling `send()`, instead of having your own domain name prepended automatically. If, in the sending application, you call

```
sendingConnection.send("receivedomain.com:mychannel", "ping");
```

the `ping()` function is called on the receiving application's local connection delegate. On the receiving side, don't specify the domain in `connect()`. It is added on automatically.

This method doesn't require changes to the receiving application, but it does require that you specify the receiving application's domain exactly. If this changes, you need to change the channel name in the sending application.

Removing the Domain from the Channel Name

If you start the channel name with an underscore (_), Flash Player doesn't automatically prepend the domain name. This means that the channel is open to any receiver on any domain. However, you're not out of the woods yet. This merely allows the connection name to be unchained from the domain.

```
//in the sending application hosted on senddomain.com
sendingConnection.send("_mychannel", "ping");

//in the receiving application hosted on receivedomain.com
listeningConnection.connect("_mychannel");
```

In the preceding code, the sending and receiving applications could be on any domain, and they would still have the opportunity to make a local connection.

Allowing Cross-Domain Local Connections from the Receiving Application

If you use a domain-free channel name (one that begins in an underscore) and communicate between applications loaded from different domains, you still have to explicitly permit the interaction. The receiving application has to approve sending applications by domain; you do so by using `allowDomain()` and `allowInsecureDomain()` on the receiving application's `LocalConnection` instance. These work the same way as `Security.allowDomain()` and `Security.allowInsecureDomain()` but apply specifically to the local connection.

```
//in the sending application hosted on senddomain.com
sendingConnection.send("_mychannel", "ping");
```

```
//in the receiving application hosted on receivedomain.com
listeningConnection.allowDomain("senddomain.com");
listeningConnection.connect("_mychannel");
```

In this example, even though the channel name is not domain specific, local connections are accepted only from applications hosted on `senddomain.com`.

If you want, you can let applications from any domain establish connections by allowing the wildcard domain.

```
listeningConnection.allowDomain("*");
listeningConnection.connect("_mychannel");
```

The `localhost` domain is reserved for applications loaded from the local filesystem; AIR applications loaded locally have their own schema based on the application ID. You can also use these methods to accept multiple domains by passing as many parameters as you like.

You can find some diagrams of the multiple ways to connect locally between cross-domain applications in the AS3LR page on `LocalConnection`.

Example: Following the Mouse

Let's flesh out a full example where the sending application sends the position of the mouse, and the connected listening application mirrors this location with an arrow.

First, the sending application is shown in Example 44-1.

EXAMPLE 44-1 `http://actionscriptbible.com/ch44/ex1`

Sending the Mouse Position

```
package {
  import com.actionscriptbible.Example;
  import flash.events.MouseEvent;
  import flash.events.StatusEvent;
  import flash.net.LocalConnection;
  public class ch44ex1 extends Example {
    protected var sendingConnection:LocalConnection;
    public function ch44ex1() {
      //sending application.
      sendingConnection = new LocalConnection();
      sendingConnection.addEventListener(StatusEvent.STATUS, onSendStatus);
      stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    }
    protected function onMouseMove(event:MouseEvent):void {
      sendingConnection.send("_mouseChannel", "mouseMoved",
                              event.stageX, event.stageY);
    }
```

EXAMPLE 44-1 *(continued)*

```
      protected function onSendStatus(event:StatusEvent):void {
        if (event.level == "error") {
          trace("ERROR: local connection failed to send the message.");
        }
      }
    }
  }
```

The receiving application is shown in Example 44-2.

EXAMPLE 44-2 `http://actionscriptbible.com/ch44/ex2`

Receiving the Mouse Position

```
package {
  import flash.display.*;
  import flash.net.LocalConnection;
  public class ch44ex2 extends Sprite {
    protected var arrow:Shape;
    protected var listeningConnection:LocalConnection;
    public function ch44ex2() {
      //listening application.
      makeArrow();

      listeningConnection = new LocalConnection();
      listeningConnection.allowDomain("*");
      listeningConnection.client = {mouseMoved: mouseMoved};
      listeningConnection.connect("_mouseChannel");
    }
    protected function mouseMoved(newX:Number, newY:Number):void {
      arrow.x = newX;
      arrow.y = newY;
    }
    protected function makeArrow():void {
      var SIZE:Number = 20, ARROW_SIZE:Number = 9, LINE_WIDTH:Number = 4;
      arrow = new Shape();
      arrow.graphics.lineStyle(LINE_WIDTH, 0, 1, false,
        LineScaleMode.NORMAL, CapsStyle.NONE, JointStyle.MITER, ARROW_SIZE);
      arrow.graphics.moveTo(-SIZE, 0);
      arrow.graphics.lineTo(SIZE-LINE_WIDTH/2, 0);

      arrow.graphics.moveTo(SIZE-ARROW_SIZE, ARROW_SIZE);
      arrow.graphics.lineTo(SIZE, 0);
      arrow.graphics.lineTo(SIZE-ARROW_SIZE, -ARROW_SIZE);
      addChild(arrow);
    }
  }
}
```

You'll have to launch both applications and make sure they're both visible to see the effect.

Summary

- Flash applications can communicate with other Flash applications running on the same machine by using `LocalConnection`.
- Local connections are completely one-way.
- Local connections invoke functions on the receiving application, optionally passing any number of arguments and up to 40KB of data.
- Use a delegate object on the receiving application to define functions that may be called from the sending application.
- By default, the `LocalConnection` object allows connections only between applications from the same domain. However, with some changes in the sending and receiving applications, communication can occur between applications from different domains.

Index

Symbols

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

# Index

Index

U

Index

The books you read to succeed.

**Get the most out of the latest software and leading-edge technologies
with a Wiley Bible—your one-stop reference.**
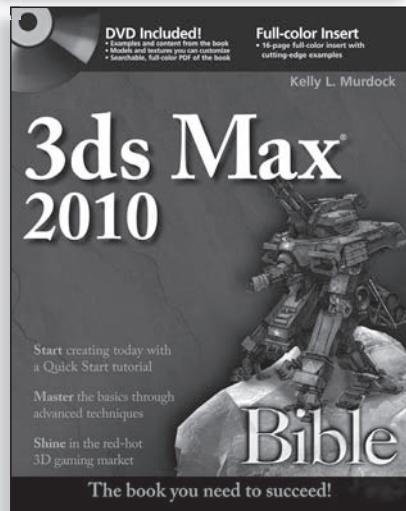
Prepare for CEH certification with this comprehensive guide

Learn how to identify security risks to networks and computers as you prepare for the Certified Ethical Hacker version 6 (CEHv6) exam. This in-depth guide thoroughly covers all exam objectives and topics, while showing you how Black Hat hackers think, helping you spot vulnerabilities in systems, and preparing you to beat the bad guys at their own game. Inside, you'll find:

Full coverage of all exam objectives in a systematic approach, so you can be confident you're getting the instruction you need for the exam

Practical hands-on exercises to reinforce critical skills

Real-world scenarios that put what you've learned in the context of actual job roles

Challenging review questions in each chapter to prepare you for exam day

Exam Essentials, a key feature in each chapter that identifies critical areas you must become proficient in before taking the exam

A handy tear card that maps every official exam objective to the corresponding chapter in the book, so you can track your exam prep objective by objective

Look inside for complete coverage of all exam objectives.

www.sybex.com

ABOUT THE AUTHOR
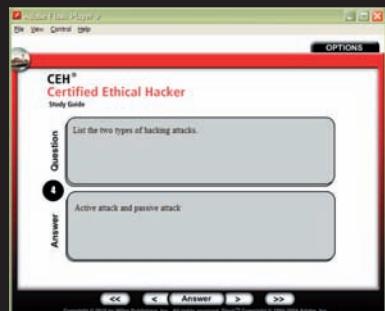
Kimberly Graves, CEH, CWSP, CWNP, CWNA, has over 15 years of IT experience. She is founder of Techsource Network Solutions, a network and security consulting organization located in the Washington, DC area. She has served as subject matter expert for several certification programs—including the Certified Wireless Network Professional (CWNP) and Intel Certified Network Engineer programs—and has developed course materials for the Department of Veteran Affairs, USAF, and the NSA.

FEATURED ON THE CD

SYBEX TEST ENGINE
Test your knowledge with advanced testing software. Includes all chapter review questions and practice exams.

ELECTRONIC FLASHCARDS
Reinforce your understanding with electronic flashcards.

Also on the CD, you'll find the entire book in searchable and printable PDF. Study anywhere, any time, and approach the exam with confidence.

CATEGORY
COMPUTERS/Certification Guides

ISBN 978-0-470-52520-3

54999

9 780470 525203

Sybex ®
An Imprint of
WILEY

$49.99 US
$59.99 CN